

NAVAL POSTGRADUATE SCHOOL

Monterey, California



19960801 075 **THESIS**

**IMPLEMENTATION AND EFFICIENCY OF STEGANO-
GRAPHIC TECHNIQUES IN BITMAPPED IMAGES AND
EMBEDDED DATA SURVIVABILITY AGAINST LOSSY
COMPRESSION SCHEMES**

by

Daniel L. Currie, III
Hannelore Campbell

March 1996

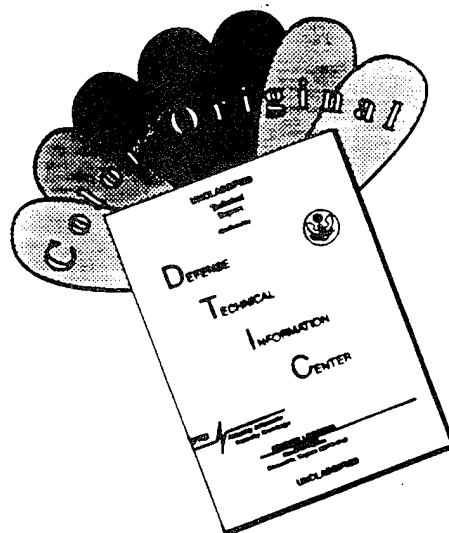
Thesis Advisors:

Cynthia E. Irvine
Harold Fredricksen

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 1

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF COLOR PAGES WHICH DO NOT REPRODUCE LEGIBLY ON BLACK AND WHITE MICROFICHE.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE March 1996		3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE IMPLEMENTATION AND EFFICIENCY OF STEGANOGRAPHIC TECHNIQUES IN BITMAPPED IMAGES AND EMBEDDED DATA SURVIVABILITY AGAINST LOSSY COMPRESSION SCHEMES			5. FUNDING NUMBERS	
6. AUTHOR(S) Currie, Daniel L. III Campbell, Hannelore				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/ MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the United States Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The term steganography is descriptive of techniques used to covertly communicate by embedding a secret message within an overt message. Such techniques can be used to hide data within digital images with little or no visible change in the perceived appearance of the image and can be exploited to covertly export sensitive information. This thesis explores the data capacity of bitmapped image files and the feasibility of devising a coding technique which can protect embedded data from the deleterious effects of lossy compression. In its simplest form, steganography in images is accomplished by replacing the least significant bits of the pixel bytes with the data to be embedded. Since images are frequently compressed for storage or transmission, it is desirable that a steganographic technique include some form of redundancy coding to counter the errors caused by lossy compression algorithms. Specifically, the Joint Photographic Expert Group (JPEG) compression algorithm, while producing only a small amount of visual distortion, introduces a relatively large number of errors in the bitmap data. These errors will effectively garble any non-coded steganographically embedded data. (continued on reverse)				
14. SUBJECT TERMS Steganography, Covert Channels, Security, Compression			15. NUMBER OF PAGES 87	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified		18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified		19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified
				20. LIMITATION OF ABSTRACT UL

Block 13 continued:

This thesis shows that, although there are numerous protocols for embedding data within pixels, the limiting factor is always the number of bits modified in each pixel. A balance must be found between the amount of data embedded and the amount of acceptable distortion. This thesis also demonstrates that, despite errors caused by compression, information can be encoded into pixel data so that it is recoverable after JPEG processing, though not with perfect accuracy.

Approved for public release; distribution is unlimited

**IMPLEMENTATION AND EFFICIENCY OF STEGANOGRAPHIC TECH-
NIQUES IN BITMAPPED IMAGES AND EMBEDDED DATA SURVIVABILITY
AGAINST LOSSY COMPRESSION SCHEMES**

Daniel L. Currie III
Lieutenant, United States Navy
B.S., West Virginia Institute of Technology, 1987
and

Hannelore Campbell
Lieutenant, United States Navy
B.S., Vanderbilt University, 1991

Submitted in partial fulfillment of the
requirements for the degree of


MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

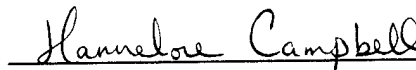
NAVAL POSTGRADUATE SCHOOL

March 1996

Authors:




Daniel L. Currie, III



Hannelore Campbell

Approved by:



Cynthia E. Irvine, Thesis Advisor



Harold Fredricksen, Thesis Advisor



Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

The term steganography is descriptive of techniques used to covertly communicate by embedding a secret message within an overt message. Such techniques can be used to hide data within digital images with little or no visible change in the perceived appearance of the image and can be exploited to covertly export sensitive information. This thesis explores the data capacity of bitmapped image files and the feasibility of devising a coding technique which can protect embedded data from the deleterious effects of lossy compression.

In its simplest form, steganography in images is accomplished by replacing the least significant bits of the pixel bytes with the data to be embedded. Since images are frequently compressed for storage or transmission, it is desirable that a steganographic technique include some form of redundancy coding to counter the errors caused by lossy compression algorithms. Specifically, the Joint Photographic Expert Group (JPEG) compression algorithm, while producing only a small amount of visual distortion, introduces a relatively large number of errors in the bitmap data. These errors will effectively garble any non-coded steganographically embedded data.

This thesis shows that, although there are numerous protocols for embedding data within pixels, the limiting factor is always the number of bits modified in each pixel. A balance must be found between the amount of data embedded and the amount of acceptable distortion. This thesis also demonstrates that, despite errors caused by compression, information can be encoded into pixel data so that it is recoverable after JPEG processing, though not with perfect accuracy.

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	WHAT IS STEGANOGRAPHY?	1
B.	PUBLISHED WORK ON STEGANOGRAPHY	1
C.	USES OF STEGANOGRAPHY	3
1.	Subliminal Communication	3
2.	Integrity and Authentication	3
3.	Illicit Exfiltration	4
D.	SCOPE OF THESIS	5
II.	STEGANOGRAPHIC TECHNIQUES	7
A.	WRAPPER FILES	7
B.	IMAGE STORAGE	8
C.	PROTOCOL EVENT SELECTION	10
D.	MESSAGE LOCATION	10
E.	COMPARISON-BASED BIT INSERTION PROTOCOLS [10]	11
1.	Bit Inversion	12
2.	Bit Insertion	13
3.	Bit Deletion	14
4.	Flag Bit	15
5.	Threshold Bits	15
F.	MESSAGE BIT INDEPENDENT INSERTION PROTOCOLS	16
1.	Direct Bit Replacement	17
2.	Neighbor Parity	17
III.	STEGANOGRAPHIC DATA CAPACITY	19
A.	DATA EMBEDDING DENSITY	19
B.	THEORETICAL LIMITS	19
C.	PERCEPTUAL LIMITS	19
1.	Pixel Structure	20
2.	Content and Texture of Images	20
3.	Hidden Data Location Within an Image	23
IV.	EMBEDDED MESSAGE SURVIVABILITY	25
A.	IMAGE PROCESSING ISSUES	25
B.	JPEG COMPRESSION	25
C.	JPEG EFFECTS ON IMAGE DATA	26
1.	Effects on Pixels	27
2.	Error Distribution	28
D.	ERROR CORRECTING SCHEMES	29
E.	AN EFFECTIVE CODING SCHEME	31
V.	DISCUSSION AND CONCLUSIONS	33
A.	STEGANOGRAPHY FOR ILLICIT EXFILTRATION	33
B.	STEGANOGRAPHIC APPLICATIONS	34
VI.	SUGGESTIONS FOR FURTHER RESEARCH	35

A.	AN UNEXPLORED DISCIPLINE	35
1.	Detecting Steganography in Image Files	35
2.	How Widespread is the Use of Steganography?	36
3.	Steganography on the World Wide Web	36
4.	Steganography in Printed Media	36
5.	Anti-Steganography Measures	36
	LIST OF REFERENCES	37
	APPENDIX: SOURCE CODE	39
A.	GENERIC STEGO EMBEDDING PROGRAM	39
B.	GENERIC STEGO EXTRACTION PROGRAM	45
C.	IMAGE FILE COMPARISON PROGRAM	56
D.	RGB VECTOR ENCODING STEGO PROGRAM	61
E.	RGB VECTOR STEGO EXTRACTION PROGRAM	68
F.	STEGO FUNCTIONS HEADER FILE	72
	INITIAL DISTRIBUTION LIST	77

I. INTRODUCTION

A. WHAT IS STEGANOGRAPHY?

Steganography is closely related to cryptography. Unlike cryptography where communication is overt, but the content is secret, steganography is the science of secretly communicating by means of concealing information within some data medium. It encompasses a large variety of techniques and protocols. Steganography can range from a simple modification of English text by modulating active and passive voices to utilizing the least significant bits in digital image pixels to unobtrusively embed additional information. Any steganographic technique will necessarily cause some distortion or modification of the original data. The key to successful steganography is to ensure the distortion caused by the hidden data is undetectable visually by either a human observer who knows the data is there or one who does not.

The term steganography was coined in 1499 by Johannes Trithemius (1462-1516) who used apparently meaningful prayers to hide covert messages [7]. Although it has been used along with cryptography for centuries in secret communications, until recently, little effort has been made to formally study or define steganography. Interest has been building, however, as evidenced by an increase in the amount of discussion of steganography in the cryptographic and computer security communities.

B. PUBLISHED WORK ON STEGANOGRAPHY

A search of the literature on and relating to steganography yields very few works exclusively discussing the subject. Steganography is usually addressed as an adjunct or offshoot of cryptography. It has rarely been discussed as a science in its own right. This is probably because, until the advent of high volume/high speed data communications, the threat of covert communication was relatively small given the usually small bandwidth provided by most steganographic techniques.

One of the earliest known published treatises on steganography was the six volume “Steganographia” by Trithemius. Trithemius described simple vowel-consonant substitutions and systems where certain letters in nonsense words comprise the hidden text. His best known invention is the “Ave Maria” in which each of many carefully selected words was used to represent a plaintext letter. The words were chosen such that when combined they made sense and appeared to be an ordinary prayer [7].

“Steganographia” was controversial due to its perceived connection with magic and the occult. Although written in 1499, it was not published until 1606 and the controversy surrounding it caused it to be banned in 1609 by the Roman Catholic Church. During the 200 years it was banned, “Steganographia” was reprinted several times and was the subject of extensive written debate by scholars.

Not until the twentieth century did steganography become recognized as a significant threat to information security. In [7] Kahn discusses the threat of covert transmission of intelligence within legitimate civilian communications during wartime. During World War II, concern over such communications led to censorship of pictures, chess boards, crossword puzzles, newspaper clippings, etc.

More recently, there has been a huge increase in large scale electronic storage of classified military data. When data is declassified or downgraded, there is a threat of covert channels being used to move data from one classification domain to another. In [9] Kurak and McHugh showed that it is a simple matter to hide information within digital images and then later extract it.

Utilizing a subliminal channel for transmission of authentication data is the subject of *The Prisoners’ Problem and the Subliminal Channel* [12]. Although referred to as subliminal channel communication by Simmons rather than steganography, authentication of covertly transmitted data is an important aspect of steganography. If a third party discovers the steganograph being used, there is a risk of spoofing.

C. USES OF STEGANOGRAPHY

Steganography is often thought of only as a tool for a malicious user to subvert a security policy, but there are actually three fundamental classes of steganography employment: subliminal communication, integrity and authentication, and illicit exfiltration of data. It is important to note that subliminal communication is not necessarily malicious or illicit. We define subliminal communication as simply a transfer of data in a manner which conceals the fact that data other than the overtly communicated information is being transmitted. Illicit exfiltration, on the other hand, is a deliberate violation of a security policy.

1. Subliminal Communication

Secretly transferring data is accomplished by using a technique to hide the data within another set of data called a wrapper. This wrapper may be virtually any type of file or data stream. A covert channel is one in which communication takes place by means that are not normally used for communication [5]. A steganographic channel is distinct from a covert channel in that it is concomitant with overt communications. The variety of steganographic channels is limited only by the creativity of the communicator. Some methods of concealing data include:

- Modulating word or line spacing in a text document
- Using the first letter of each word or paragraph as a letter of the message
- Modulating the tense or voice of verbs
- Using the least significant bit(s) (LSB) of each pixel in an image file
- Using the LSB(s) in a digitized sound file

2. Integrity and Authentication

Integrity of data is another important aspect of computer system security. Steganographic techniques can be used to embed a seal within an image file in such a way that the data cannot be modified by even a single bit without detection. This provides both authentication and tamper-proofing of the image. Steganography has the advantage that the

presence of embedded data is usually not detectable and cannot be altered or removed without knowledge of the exact technique used.

In [14], Walton describes a technique which guarantees the integrity of image files. This method employs an algorithm which visits certain, randomly chosen pixels, modifying them to produce a desired checksum value. The seal key is the order in which the pixels are visited. This “random walk” sealing method ensures integrity since “...an interloper cannot tell if an image has been sealed, has no way of finding the unchanged LSBs, and cannot blanket your image with all possible seals.”

Documents and images are tagged for a variety of reasons from enforcing copyright ownership to archival marking of data. Steganographic techniques provide an unobtrusive way to tag a file without lengthening the file or subjecting the tag to possible alteration or removal. AT&T explored a tagging system in which subtle changes in line spacing in postscript files were used to encode a serial number. This technique proved to be effective even when the document was photocopied numerous times. [2]

A successful steganographic technique for authentication and integrity will necessarily rely on protecting the steganographic method and any keys used to encrypt the hidden data. Additionally, once data has been steganographically embedded within a file, care must be taken to keep the original file separate from the altered file. Allowing comparison of the two files undermines the secrecy of the steganographic channel's existence.

3. Illicit Exfiltration

Of the many means of illicit exfiltration of data, steganography is perhaps the most insidious in that it requires a sophisticated user to implement and can provide a high bandwidth communication channel. It is an especially grave threat in an environment where data files are being downgraded from a high security class to a lower one. The malicious user can embed classified data within a wrapper file of equal classification which he expects to be downgraded. The data can then be extracted when the wrapper file is moved to a lower security classification domain.

D. SCOPE OF THESIS

This thesis is intended to be expository on the subject of steganography, to survey steganographic techniques and their efficiency, and to investigate the feasibility of devising a robust form of steganography that can survive lossy compression. Our ultimate goal is to highlight the threat of steganography to system security and to demonstrate a steganographic technique which could be used to thwart a multi-level system security policy in an image storage, processing, and downgrading system which relies on a trusted subject.

II. STEGANOGRAPHIC TECHNIQUES

A. WRAPPER FILES

A wrapper is the medium into which a message (hidden data) is embedded. As with encryption, steganography can be performed on a block of data or on a data stream. For the purposes of this thesis we will be concerned with block steganography carried out on data files rather than stream steganography. A variety of data files are commonly used in computer systems. Each type of file has characteristics which must be taken into account when devising a steganographic technique. In general, files in which the data is stored with a far greater accuracy than necessary for the data's use and display are suitable for steganography. Image, Postscript, and audio files are among those which fall into this category, while text, database, and executable files do not.

A text file contains a relatively low data density. Certain aspects of the file, such as the bits which comprise the characters, are off limits to steganography because any changes cause obvious damage to the file. Other aspects, such as character, word, or line spacing, can be subtly modified to hide data. These techniques yield a low bandwidth, but are useful for document tagging or any other purpose which requires a small amount of hidden data.

With the increased use of multimedia in applications, sound files are much more common and provide an ideal medium for steganography. Due to the imprecise nature of human hearing, modifications to a sound file by a steganographic technique would have to be relatively large to be detected by a human observer. With a properly designed steganographic technique, even an informed observer listening to the modified and unmodified sound files in quick succession would have difficulty hearing a change. Consequently, a sound file can hold a much larger message than can be embedded in a text file.

By far the most readily available and suitable wrapper files are image files. Image files are ubiquitous in a large number of applications. As with sound files, image files have a high data density and small changes can be made without detection.

B. IMAGE STORAGE

The format in which images are created and stored varies with the application. The image can be as simple as a bi-level format where each pixel is either on or off corresponding to bit value of 1 or 0. A better quality image can be made by using 8 bits for each pixel, allowing 256 shades of gray per pixel. Using 24 bits for a pixel to encode a color value yields an even more accurate and realistic image. Clearly any number of bits and pixel encoding schemes can be used to provide any desired level of detail. In general, as the number of bits used to represent a discrete component of a file (in this case a pixel) increases, the file's utility as a wrapper increases.

Due to their gross granularity, bi-level images are difficult to utilize as wrapper files. A steganographic technique would introduce noise into the image which would likely cause noticeable degradation. This effect is mitigated in complex or noisy images. In sparsely detailed images, however, any changes will probably be easily spotted.

Grey scale images are much more suitable for hiding messages. Altering the low order bit of the eight bits used to represent a pixel causes only a $1/256$, or 0.39%, change in the shade of grey. This small change is undetectable by the human eye. Furthermore, altering the four low order bits induces at most a $16/256$, or 6.25%, change.

The percentages above represent a worst case scenario. When a message bit is embedded into the wrapper file, the target bit in the wrapper file will be changed only if it differs from the message bit. The likelihood that the target bit will be changed is 0.5.

Therefore, embedding n bits will result in an average of $n/2$ bits being modified in the wrapper file.

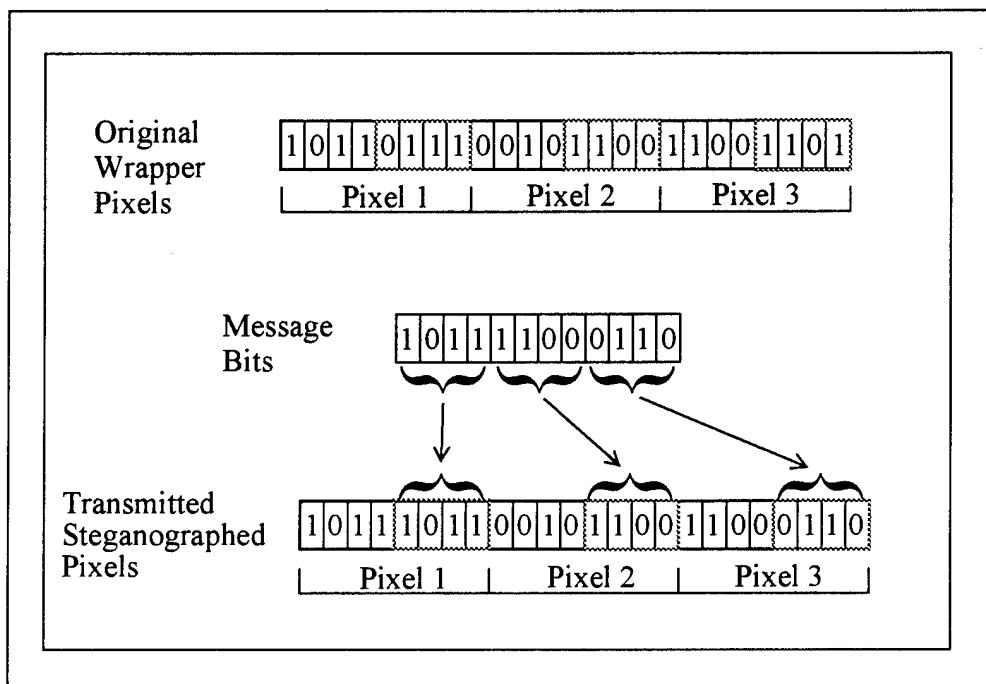


Figure 1: Direct Insertion of Message Bits

Figure 1 demonstrates how a message can be embedded into a wrapper file composed of 8 bit pixels representing 256 shades of grey. The steganographic technique replaces the four low order bits of each pixel with the data bits. In the first pixel only two bits are changed, resulting in a $4/256$, or 1.56%, change in its shade of grey. The message bits embedded in the second pixel matched the pixel's original bits, therefore no change was made.

Color images, specifically 24 bit color images, are extremely common. One can easily extrapolate from the above example and see that a 24 bit pixel is capable of holding much more embedded data with only a small change in the appearance of the pixel. This thesis will focus on this type of file. The amount of data which can be embedded in a 24 bit color

image file and the degree of the resulting distortion of the image as a whole will be investigated.

C. PROTOCOL EVENT SELECTION

When applying a steganographic technique to a wrapper file there must be a protocol for the selection of the specific points in the file at which the message bits will be insinuated. This is known as protocol event selection (PES). In the context of image files the PES algorithm will select pixels in which data will be hidden.

The PES algorithm is an integral part of any steganographic technique. For our purposes, it involves selecting pixels in an image file viewed as a linear arrangement of pixels numbered from 1 to $W \times H$, where W is the width in pixels of the image and H is the height. It can be defined as a mathematical function which generates numbers corresponding to the numbers of the pixels to be used. A simple example would be to use every pixel p where $p \bmod 2$ is equal to 0 (i.e. every even numbered pixel). Any function is suitable as long as it selects enough pixels to embed the desired number of bits.

Another PES method is based on a predetermined key which provides a map of the order in which pixels are used. This key may be designed so as to produce a pseudo-random "walk" from pixel to pixel [14]. The walk, however it is generated, must not visit the same pixel twice.

When incorporating a predetermined key scheme into a steganographic protocol, provision must be made for embedding the key in the image along with the hidden data or passing the key separately. This PES method is also very sensitive to errors in the key since any key error would make complete recovery of the embedded data nearly impossible.

D. MESSAGE LOCATION

Once a PES method is established, the location within the image where the message will be hidden must be determined. Commonly, the entire image is used in order to maximize the amount of data which can be hidden. Using the entire image also spreads the message over the image so as to reduce the density of the hidden data and thereby reduces

the perceived distortion of the image. Nevertheless, there are several techniques and reasons for localizing the message within an image.

The goal of steganography is to hide data in such a way that it is not detectable by the human eye. One way of doing this is to put the hidden data in an area where it is unlikely to be noticed. Most pictures have a subject which is the center of attention to which the eye is naturally drawn. Lacking a particular subject or area of interest, the eye will be drawn to the center of the picture. Clearly, these areas should be avoided by a steganographic technique. Another consideration for the location of hidden data is the texture, or “busyness,” of different areas of the picture. For example, in a photograph of a forest with clear blue sky overhead, distortion would be more easily detected in the sky area than in the forest area.

Setting aside the issue of covertness, it is often desirable to provide redundancy in the hidden data to counter the effects of errors introduced into the image file. Figure 2 shows possible arrangements of redundant hidden data.

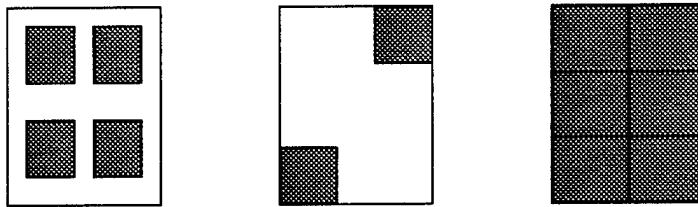


Figure 2: (Shaded areas indicate hidden data)

Incorporating redundancy in this way provides some protection against errors introduced in transmission, processing, and compression. It can also allow for recovery of the hidden data when the entire image file is not available.

E. COMPARISON-BASED BIT INSERTION PROTOCOLS [10]

The actual insinuation of message bits can be accomplished by several protocols. The PES technique can be viewed as a means of moving a pointer to an insertion point within

the bits of the wrapper file. Having selected an insertion point, an insertion protocol is used to place a message bit in the wrapper file. The insertion protocol can involve a replacement of the insertion point bit or its neighbors, or insertion of the message bit between wrapper bits near the insertion point. Inserting message bits between wrapper bits has the effect of shifting all the wrapper bits after the insertion point. This will cause undesirable effects in some types of wrapper files. Specifically, if bits in an image file are shifted without regard to the pixels (i.e. the image file is treated as a continuous stream of bits rather than a series of discrete pixels) then insertion of a few bits will essentially randomize the image.

1. Bit Inversion

The insertion point bit is specially selected to encode the message bit. If the message bit is 1 then the insertion point bit must be a 0 and is inverted to represent the message bit. Likewise, if the message bit is a 0 then the insertion point bit must be a 1.

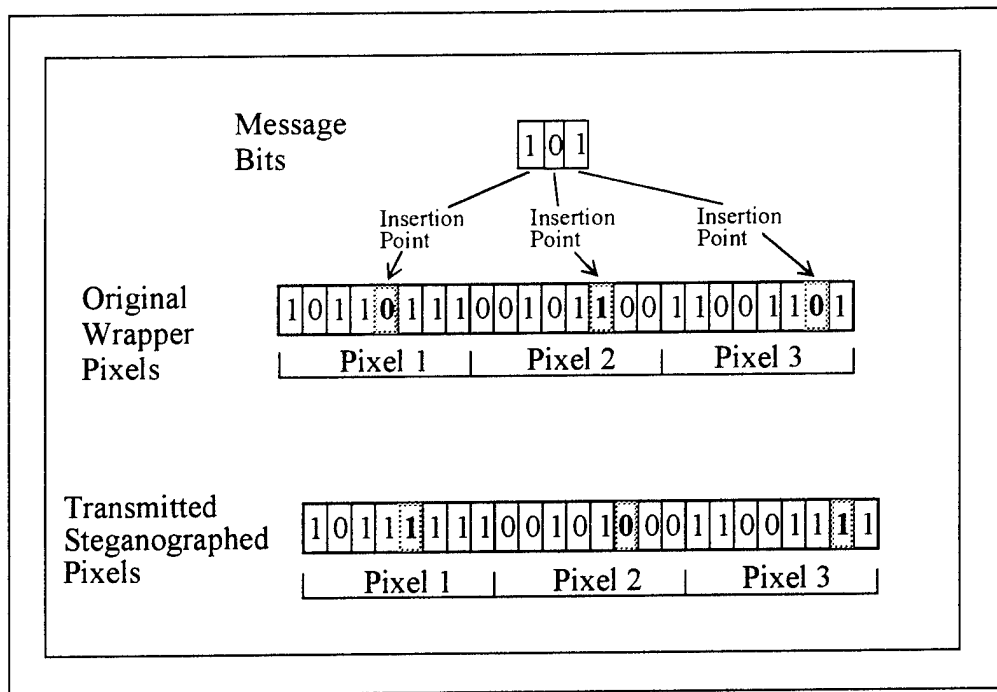


Figure 3: Bit Inversion

This technique requires that the original wrapper be compared to the stegotext to recover the message. When comparing the files, every differing bit and its value is noted.

2. Bit Insertion

The insertion point bit is chosen to be the opposite value of the message bit. The message bit is then inserted immediately prior to the insertion point bit.

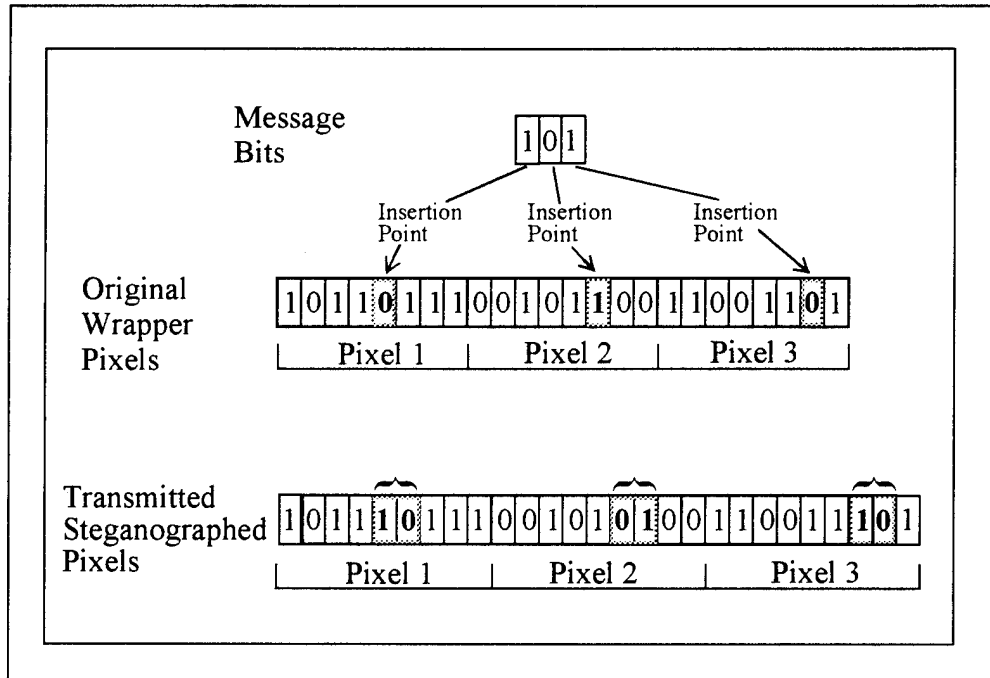


Figure 4: Bit Insertion

The message is extracted by comparing the wrapper with the stegotext and noting where a bit differs and its value. In this technique an extra bit is inserted in the wrapper so when the message bit is extracted, the bits to the right must be shifted left to regain alignment.

3. Bit Deletion

In this case, the insertion point is chosen as a pair of bits; either a 01 or 10 depending on the message bit. If a 1 is to be inserted then a 10 pair is chosen and the 1 of the pair is deleted. Similarly if a 0 is to be inserted, a 01 pair is chosen and the 0 of the pair is deleted.

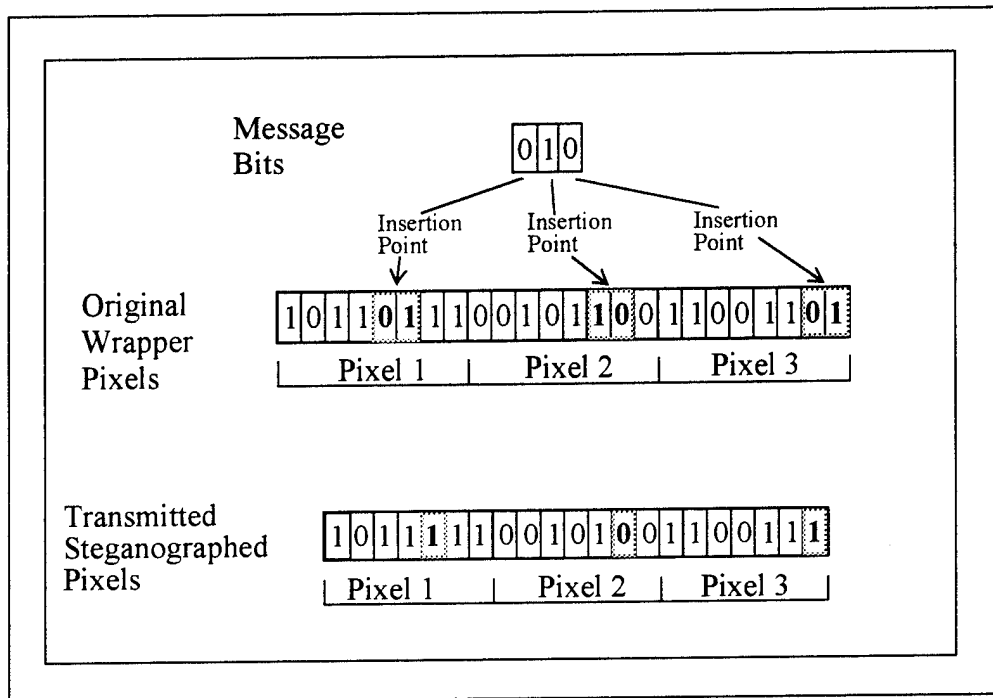


Figure 5: Bit Deletion

Again, the message is extracted by noting the difference between the wrapper and the stegotext. A right shift is necessary to regain alignment and continue with the comparison.

4. Flag Bit

With this technique, a wrapper bit equal to the message bit is selected. The bit preceding the selected wrapper bit is inverted and acts as a flag for the message bit.

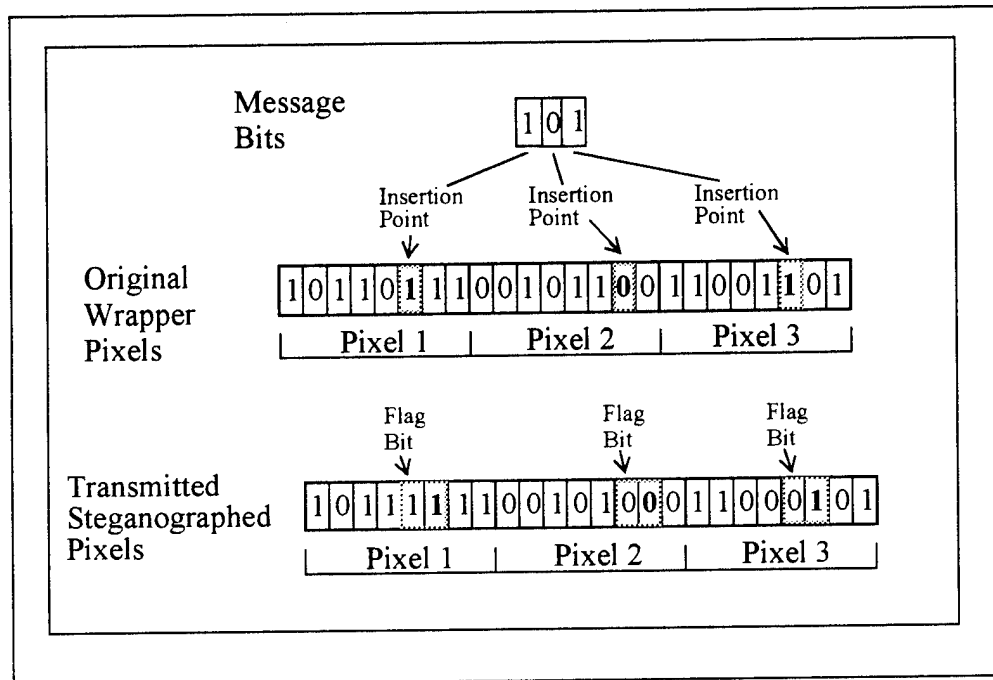


Figure 6: Flat Bit

The message is extracted by comparing the wrapper to the stegotext and noting the bit after each comparison failure.

5. Threshold Bits

Like the flag bit protocol, an inverted bit acts as a flag for the message bit. In this case however a certain odd number n of bits, immediately following the flag bit are used to encode the value of the message bit. If the number of 1's in the n bits is greater than the

number of 0's then the message bit is 1. Similarly, if there are more 0's than 1's then the message bit is 0. This protocol allows for error correction.

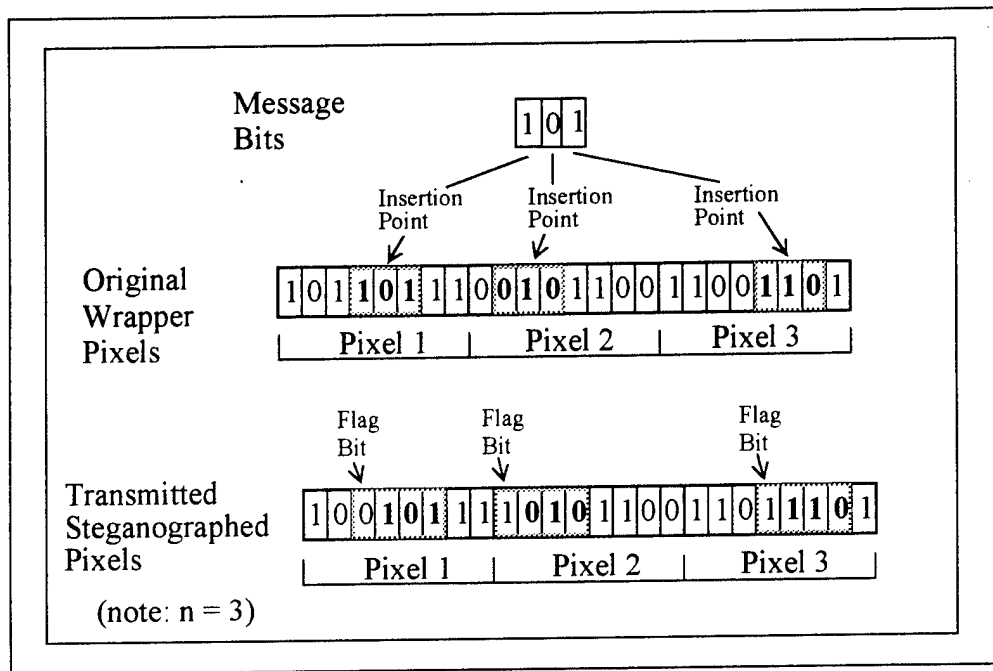


Figure 7: Threshold Bits

F. MESSAGE BIT INDEPENDENT INSERTION PROTOCOLS

The protocols in the previous section require the receiver to have both the original wrapper file and the stegotext to extract the message. This is because the insertion point bit(s) are selected based on the value of the message bit. Therefore the only way for the receiver to identify the insertion points is to compare the wrapper with the stegotext. This is entirely different from the key- or algorithm-based protocol event selection techniques mentioned earlier.

Frequently, it is inconvenient or impossible for the receiver to have the original wrapper file. For example, in an image downgrading scenario, it is possible for a misfeasor to embed classified data in an image which he expects to be downgraded. Once the wrapper image is downgraded, the original wrapper file is no longer available to be used for comparison in an extraction algorithm. For this reason, when designing a steganographic

application it is preferable to use a protocol which does not rely on the availability of the original wrapper file. The following are examples of this type of protocol.

1. Direct Bit Replacement

In this protocol, the insertion point bit is simply replaced by the message bit. This protocol was demonstrated in Figure 1.

2. Neighbor Parity

As with the threshold bit protocol discussed earlier, this protocol examines n neighboring bits and modifies the insertion point bit to achieve a desired parity.

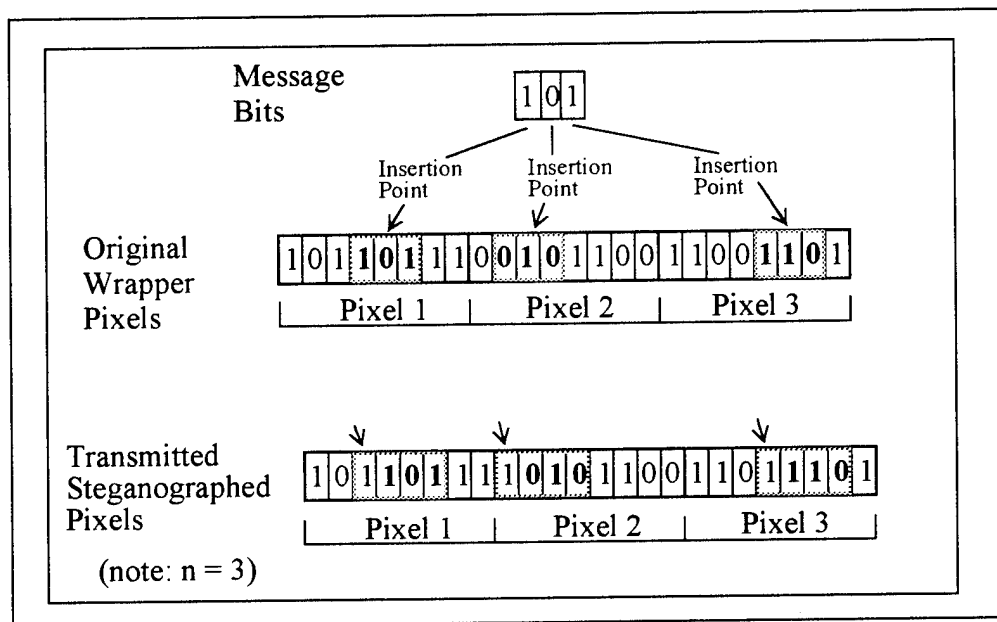


Figure 8: Neighbor Parity

For this example, odd parity represents 1 and even parity 0. Clearly, this technique requires the PES to maintain at least n bit spacing between insertion points. The overall effect of this technique is to essentially randomize the way message bits are represented. Sometimes a message bit of 1 will be represented at the insertion point as a 0 and sometimes as a 1.

III. STEGANOGRAPHIC DATA CAPACITY

A. DATA EMBEDDING DENSITY

The relationship between the size of an image file and the amount of data which can be hidden in it is very straightforward. The *steganographic data density*, defined as the number of data bits per image byte, directly determines the total amount of data which can be hidden. The general formula for the data capacity in bytes is:

$$\text{data capacity} = ((W * H * \text{bytes per pixel}) * D) / 8 \text{ bits per byte},$$

where W and H are the image width and height in pixels and D is the number of data bits embedded per pixel.

B. THEORETICAL LIMITS

As described above, the number of bytes of data which can be hidden is dependent only on the size of the image file and the density at which data is embedded. The density is, in turn, determined by the acceptable degree of distortion as will be described in the next section.

The amount of information which can be stored can be improved through the use of compression. From the point of view of the steganographic algorithm the data is a stream of bytes. There is no reason why a stream of bytes representing ASCII characters cannot be embedded in a compressed form thereby increasing the total amount of information which can be stored in a given image.

C. PERCEPTUAL LIMITS

In deciding on a hidden data density one must strike a balance between the amount of image degradation and the desire to maximize the amount of data which can be stored in an image. Increasing the data density increases the amount of degradation. The trick is to find the data density threshold at which an uninformed observer may notice that the image is fuzzy. Three factors which influence this are the pixel structure, the content and texture of the image, and the hidden data location within the image.

1. Pixel Structure

In any image file format which uses less than 24 bits per pixel, some provision for a color map must be made in order for the file to be displayed on a color monitor. This is due to the necessity of converting the pixel value to a 24 bit value for the display driver. The most common pixel structure of this kind is 8 bits. Since an 8 bit value can only represent 256 colors, a color map is necessary to map the 256 colors to the 16,777,216 possible colors.

Any attempt to embed data in an 8 bit pixel value will result in that pixel value pointing to an entirely different color value in the color map. A color map in an 8 bit image file is the result of some kind of quantization of the colors present in the image, so changing a pixel value by even a small amount may cause a drastic change in the color of the pixel [3].

A 24 bit image file format is ideal for steganographic purposes. A 24 bit pixel is composed of three 8 bit color channels: red, green, and blue. The number of possible colors is very large and it is possible to change several bits without making a noticeable change in the color. This is due to two factors. First, although the human eye can distinguish a large number of colors under ideal conditions, viewing an altered image on a monitor without the benefit of a reference image makes detecting small color changes unlikely. Second, while the pixel value may be 24 bits, most personal computers and workstations do not display 24 bits of color on the monitor. The 24 bit pixels are usually mapped internally to 4, 8, or 16 bits for the display. Further, every monitor will have small differences in color saturation, brightness, contrast, etc.

2. Content and Texture of Images

The characteristics of the image itself is another factor which must be considered when applying a steganographic technique. The overall variety of colors, shapes, and textures and their relative sizes and dispositions can affect the detectability of embedded data. For example, in Figure 9, the seagull is seen against a sky which is composed of only a few shades of blue. Color distortion introduced in this mostly blue field will appear as specks

of darker or lighter blue, or even a different color, which contrasts with the blue around it as seen in Figure 10. (Distortion has been enhanced for illustration.)

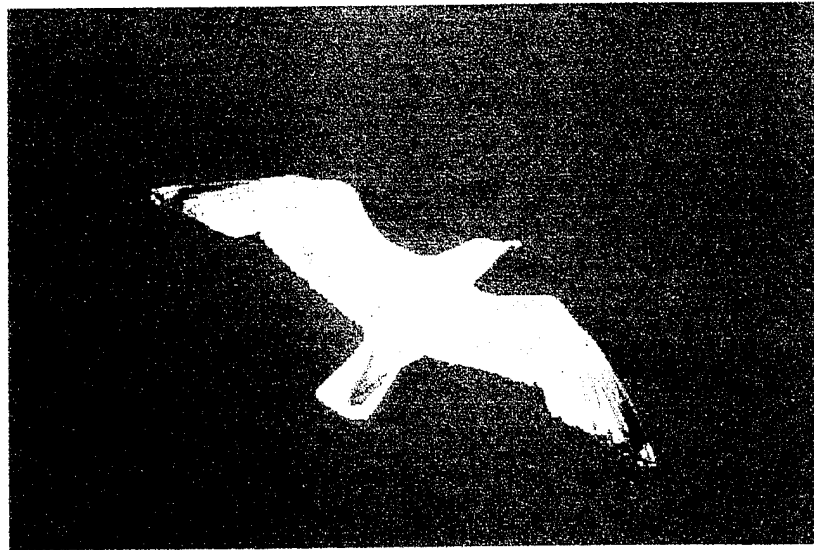


Figure 9: Seagull, 730x480, 24 bit pixels [8]

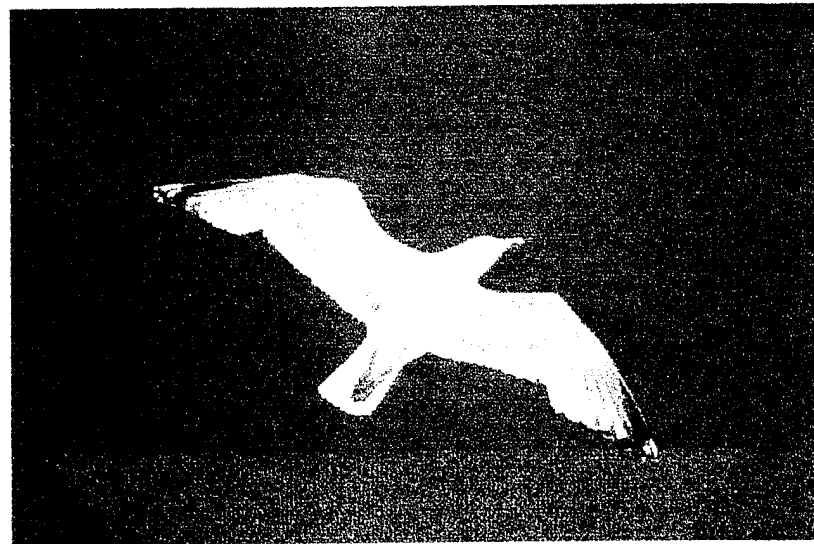


Figure 10: Seagull with 136,608 bytes of data embedded

A good choice for an image to be used as a wrapper would be one which is very busy, i.e. contains many different shapes, colors and textures. In this type of picture, the

steganographic distortion will be lost in the profusion of detail. This is demonstrated in Figure 11 and Figure 12. Figures [10] and [12] contain the same amount of embedded data, but it is much less visible in Figure 12.



Figure 11: Glasses, 730x480, 24 bit pixels [8]



Figure 12: Glasses with 136,608 bytes of data embedded

3. Hidden Data Location Within an Image

The steganographic data capacity of an image is a function of the embedding density. In the case where less than the maximum amount of data is embedded, care must be taken to avoid creating a line of demarcation between the area containing hidden data and the portion of the image which has not be tampered with. In Figure 10 the hidden data is obviously located in the bottom portion of the image.

A solution to this problem is to compute n , the number of bytes in the image divided by the number of image bytes necessary to hold the embedded data. Then, in the embedding algorithm, use every n th image byte. This will spread the hidden data over the entire image, reducing its detectability. Spreading out the data in Figure 10 results in Figure 13; clearly, a significant improvement.

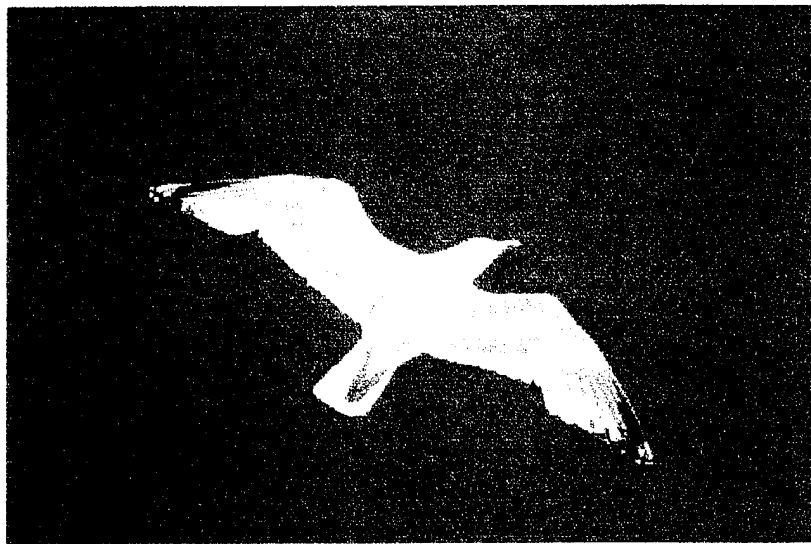


Figure 13: Seagull with embedded data spread out

IV. EMBEDDED MESSAGE SURVIVABILITY

A. IMAGE PROCESSING ISSUES

Both newly created and existing images often undergo modification as a result of some form of processing. Images can be compressed using lossy compression techniques, can be digitally enhanced, or they can have other images or graphics overlaid on them. All forms of 'invasive' processing are a threat to any steganographically embedded message. Alteration of any of the pixels containing message bits can garble the message. To enhance message survivability, some form of error correction is necessary.

The most frequently encountered source of image degradation is lossy compression. In the next two sections we will examine the loss caused by the Joint Photographic Experts Group (JPEG) compression standard and the error correction required to compensate for it. [6]

B. JPEG COMPRESSION

The JPEG compression standard has been developed to provide efficient, flexible compression tools. JPEG has four modes of operation designed to support a variety of continuous-tone image applications. Most applications utilize the Baseline sequential coder/decoder which is very effective and is sufficient for many applications. [13]

JPEG works in several steps. First the image pixels are transformed into a luminance/chrominance color space [11] and then the chrominance component is downsampled to reduce the volume of data. This downsampling is possible because the human eye is much more sensitive to luminance changes than it is to chrominance changes. Next, the pixel values are grouped into 8x8 blocks which are transformed using the discrete cosine transform (DCT). The DCT yields an 8x8 frequency map which contains coefficients representing the average value in the block and successively higher-frequency changes within the block. Each block then has its values divided by a quantization coefficient and the result rounded to an integer. During this quantization is where most of the loss caused

by JPEG occurs. Many of the coefficients representing higher frequencies are reduced to zero. This is acceptable since the higher frequency data that is lost will produce very little visually detectable change in the image. The reduced coefficients are then encoded using Huffman coding to further reduce the size of the data. This step is lossless. The final step in JPEG applications is to add header data giving parameters to be used by the decoder. [13]

C. JPEG EFFECTS ON IMAGE DATA

As mentioned before, embedding data in the least significant bits of image pixels is a simple steganographic technique, but it cannot survive the deleterious effects of JPEG. To investigate the possibility of employing some kind of encoding to ensure survivability of embedded data it is necessary to identify what kind of loss/corruption JPEG causes in an image and where in the image it occurs.

At first glance, the solution may seem to be to look at the compression algorithm to try to predict mathematically where changes to the original pixels will occur. This is impractical since the DCT converts the pixel values to coefficient values representing 64 basis signal amplitudes. This has the effect of spatially "smearing" the pixel bits so that the location of any particular bit is spread over all the coefficient values. Because of the complex relationship between the original pixel values and the output of the DCT, it is not feasible to trace the bits through the compression algorithm and predict their location in the compressed data.

Due to the complexity of the JPEG algorithm an empirical approach to studying its effects is called for. To study the effects of JPEG, 24 bit Windows BMP format files were compressed, decompressed, and the resulting file saved under a new filename.

The BMP file format was chosen for its simplicity and widespread acceptance for image processing applications. For the experiments, two photographs, one of a seagull and one of a pair of glasses (Figure 9 and Figure 11), were chosen for their differing amount of detail and number of colors. JPEG is sensitive to these factors. To investigate JPEG's effects, an analysis was conducted to locate and quantify the errors introduced.

1. Effects on Pixels

Table 1 below shows the results of a byte by byte comparison of the original image files and the JPEG processed versions, normalized to 100,000 bytes for each image. Here we see that the seagull picture has fewer than half as many errors in the most significant bits (MSB) as the glasses picture. While the least significant bits (LSB) have an essentially equivalent number of errors.

	MSB 8	7	6	5	4	3	2	LSB 1
Glasses	744	4032	10247	21273	33644	42327	27196	48554
Seagull	257	991	2821	7514	15039	29941	41593	46640

Table 1: Errors introduced by JPEG per bit position

Table 2 shows the Hamming distance (number of differing bits) between corresponding pixels in the original and JPEG processed files normalized to 100,000 pixels for each image. Again, the seagull picture has fewer errors.

	1-3	4-6	7-9	10-12	13-15	16-18	19-21	22-24
Glasses	15581	37135	30337	11976	2205	172	4	0
Seagull	24188	38710	17564	4631	409	43	1	0

Table 2: Hamming distances between pixels

Given the information in Table 1, it is apparent that data embedded in any or all of the lower 5 bits would be corrupted beyond recognition. Attempts to embed data in these bits and recover it after JPEG processing showed that the recovered data was completely garbled by JPEG.

2. Error Distribution

Although JPEG causes errors throughout the image, the most severe errors, that is, the pixels suffering the largest number of bit inversions, are located along edges or transients in the pixel values. Figure 14 is a pixel by pixel comparison of the original seagull picture (Figure 9) and the JPEG processed version of the same file. Each black dot indicates a pixel which has a Hamming distance of more than 8 from the corresponding original pixel.

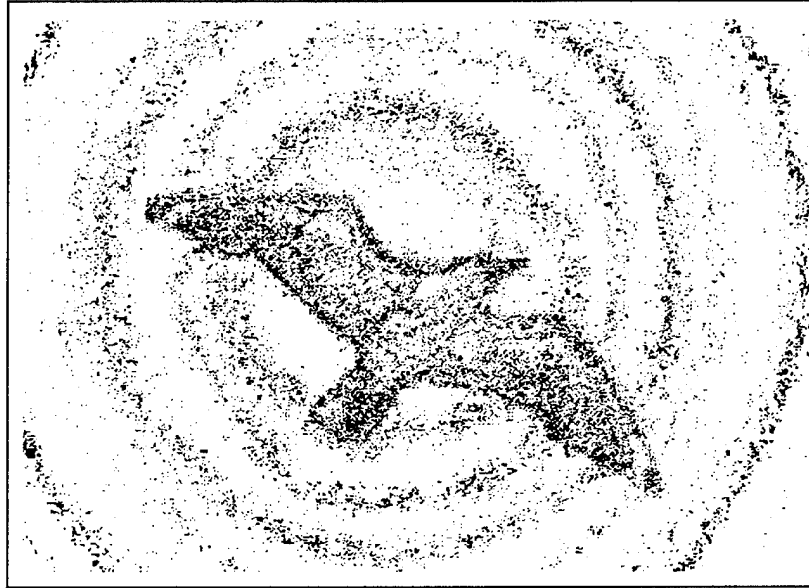


Figure 14: Map of errors of more than 8 bits per pixel

Given the nature of the JPEG errors, the problem of embedding the data and of applying an error correction scheme is challenging. The fact that the lower order bits are severely affected by JPEG indicates that data should be embedded in higher order bits, but this will cause more degradation of the image and compromise the effectiveness of the steganographic scheme. Also, since the number of errors is high and distributed across all bit positions, a robust error correcting scheme is needed to protect the hidden data from corruption. The price of robust error correction is a large overhead which must be added to the data. This reduces the total amount of information which can be hidden.

D. ERROR CORRECTING SCHEMES

Since a straightforward substitution of pixel bits with data bits proved useless, a simple coding scheme to embed one data bit per pixel byte was tried. A bit was embedded in the lower 5 bits of each byte by replacing the bits with 01000 to code a 0 and 11000 to code a 1. On decoding, any value from 00000 to 01111 would be decoded as a 0 and 10000 to 11111 as a 1. The theory was that perhaps JPEG would not change a byte value by more than 7 in an upward direction and 8 in a downward direction or, if it did, it would make drastic changes only occasionally and some kind of redundancy coding could be used to correct errors. This approach failed. JPEG is indiscriminate about the amount of change it makes to byte values and produced enough errors that the hidden data, when extracted, was unrecognizable. This is not surprising since the JPEG algorithm spreads the value of each bit over many coefficient values, some of which are reduced to zero in the quantization step.

The negative results of the first few attempts to embed data indicated that a more subtle approach to encoding was necessary. It was noticed that, in a JPEG processed image, the pixels which were changed from their original appearance were similar in color to the original. This indicates that the changes made by JPEG, to some extent, maintain the general color of the pixels. This characteristic is desirable in an algorithm for compressing digital images; the appearance of the image should change as little as possible. To attempt to take advantage of this, a new coding scheme was devised based on viewing the pixel as a point in space (Figure 15) with the three color channel values as the coordinates.

The coding scheme begins by computing the distance from the pixel to the origin (0,0,0). Then the distance is divided by a number and the remainder ($r = \text{distance} \bmod n$) is found. The pixel value is adjusted such that its modulus is changed to a number corresponding to the bit value being encoded. Qualitatively, this means that the length of the vector representing the pixel's position in three-dimensional RGB color space is

modified to encode information. Because the vector's direction is unmodified, the relative sizes of the color channel values are preserved.

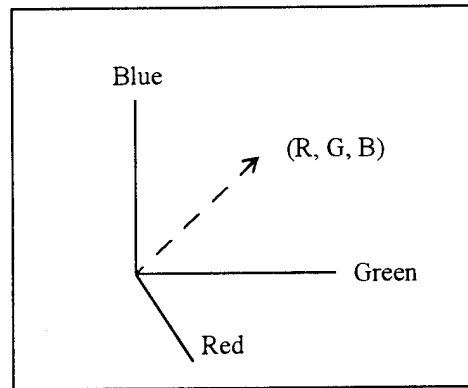


Figure 15: A pixel as a point in space

Suppose we choose an arbitrary modulus of 42. When the bit is decoded, the distance to the origin will be computed and any value from 21 to 41 will be decoded as a 1 and any value from 0 to 20 will be decoded as a 0. So we want to move the pixel to a middle value in one of these ranges to allow for error introduced by JPEG. In this case, the vector representing the pixel would have its length modified so that the modulus is 10 to code a 0 or is 31 to code a 1. It was hoped that JPEG would not change the pixel's distance from the origin by more than 10 in either direction thus allowing the hidden information to be correctly decoded.

For example, given a pixel (128, 65, 210) the distance to the origin would be computed: $d = \sqrt{128^2 + 65^2 + 210^2} = 254.38$. The value of d is rounded to the nearest integer. Next we find $r = d \bmod 42$, which is 2. If we are coding a 0 in this pixel, the amplitude of the color vector will be increased by 8 units to an ideal modulus of 10 ($d = 262$) or moved down 13 ($d = 241$) units to code a 1. Note that the maximum displacement any pixel would suffer would be 21. Simple vector arithmetic permits the modified values of the red, green, and blue components to be computed. The results of using this encoding are described in the next section.

Another similar technique is to apply coding to the luminance value of each pixel in the same way as was done to the distance from origin. The luminance, y , of a pixel is computed as $y = 0.3R + 0.6G + 0.1B$ [11], where R , G , and B are the red, green, and blue color values respectively. One drawback to this technique is that the range of luminance value is only from 0 to 255 whereas the range of the distance from origin is 0 to 441.67.

E. AN EFFECTIVE CODING SCHEME

With ordinary embedding techniques which simply replace image bits with data bits, one is forced to use an enormous amount of redundancy to achieve a suitably low error rate after JPEG compression. Also, since the lowest few bits are so badly scrambled by JPEG, higher order bits must be used which increases the visible distortion of the image. This is contrary to steganography's goal of being a covert communication channel.

The distance from the origin technique results in a lower error rate, but requires additional redundancy to reduce the error rate to a point where recognizable text can be recovered. Specifically, the average displacement by JPEG of pixels in the seagull picture [Figure 9] with respect to the origin is 2.36. Theoretically, if a modulus of 62 is assumed, the number of pixels displaced by 30 (enough to cause an error) is 3100 or 0.8678%. Given this figure and applying triple redundancy in embedding data (i.e. embed each bit three times in a row) yields an error rate of $p^3 + 3p^2(1-p) = (.008678)^3 + 3(.008678)^2(.991322) = 0.000225$ per bit, where p is the probability of a bit error. This would yield an error rate of: $1 - (1 - 0.000225)^8 = 0.001799$ per byte.

In practice, however, utilizing the encoding described in the above example will not produce the accuracy as calculated. The problem lies in the nature of the effect JPEG has on an image. When a pixel is modified to encode a bit value, the difference in its value with respect to surrounding pixels changes. JPEG is sensitive to transients between adjacent pixels, i.e. the greater the number and size of transients, the greater the number of errors introduced by JPEG. Thus, the very act of encoding data into pixels generates an increase in the severity of distortion and results in garbling of the embedded data.

This characteristic of JPEG makes it a very good anti-steganography tool, but does not preclude steganographic embedding entirely. With a high repetitive redundancy factor, some text may be embedded and extracted from a JPEG processed image. Using distance to the origin encoding and repeating each bit 5 times, approximately 30% of the text can be recovered without error.

V. DISCUSSION AND CONCLUSIONS

A. STEGANOGRAPHY FOR ILLICIT EXFILTRATION

As was seen in the previous chapters, steganography is relatively simple to implement and is capable of hiding a significant amount of information in an image file. Until recently, the very notion of digital steganography was not known or was not perceived as a threat. The increase in communication bandwidths has encouraged a manifold increase in the number of digital images being transmitted and stored. In light of this, steganography must be considered when developing a computer security policy.

Although steganography is considered by many to be a form of covert channel, in the strictest sense it is not. A covert channel uses mechanisms in a computer system for a purpose they were not intended [1]. For example, the amount of disk storage available can be modulated by a process operating at a high security level. A process at a lower security level can monitor the effects of this modulation and thereby receive information from the higher security level. In contrast, steganography can exploit a trusted subject's privileges in a multi-level system. Specifically, those privileges which allow it to write to a lower classification domain. In the context of an image downgrading system where a human operator is using a trusted subject to change the classification of images, steganography takes advantage of the operator's inability to discern or identify the distortion caused by embedded data. Since human perception is not sufficient in this case to prevent flow of data to lower classification domains, additional controls are called for.

The key to preventing steganographic exfiltration of data is a strong integrity control system. One method which may be used to protect the integrity of image files is an embedded checksum or message digest [4]. A checksum or message digest would be computed, encrypted and steganographically embedded in an image. Any tampering with the image would render the embedded integrity seal unrecoverable. (Note that encryption keys and methods may need to be protected using a high assurance security policy enforcement mechanism.)

Another method of protecting a system against steganographic exfiltration of data is to employ anti-steganography measures when handling image files. Transform based lossy compression, e.g. JPEG, is very effective in garbling embedded data, but cannot be relied upon for complete protection. It is possible for embedded data to be encoded to resist the effects of the distortion. The amount of data hidden in such a way would be small compared to the size of the image due to the large amount of error correcting data which must be added to the information being embedded.

B. STEGANOGRAPHIC APPLICATIONS

Steganography is a relatively simple concept and implementations of steganography are not difficult to develop with commonly available tools, e.g. C++, awk, shell scripts, etc. A competent programmer could write a steganographic application and an accompanying shell script to embed data to be exfiltrated in all image files available to him. This could be accomplished in batch mode to spread out the processing load this would impose on a system.

The threat of Trojan horses to system security is well documented. Any application which handles files which are suitable for steganography could contain code to hide data in the files. An example of this is JPEG applications. Since, in the normal course of its processing, JPEG introduces distortion into an image, a Trojan horse steganography algorithm would have an extra measure of covertness i.e. an observer would assume the image distortion is due entirely to JPEG processing.

VI. SUGGESTIONS FOR FURTHER RESEARCH

A. AN UNEXPLORED DISCIPLINE

This thesis explores only a small part of the science of steganography. As a new discipline, there is a great deal more research and development to do. The following sections describe areas for research which were offshoots of, or tangential to, our main objectives.

1. Detecting Steganography in Image Files

Can steganography be detected in image files? This is a difficult question. It may be possible to detect a simple steganographic technique by simply analyzing the low order bits of the image bytes. If the steganographic algorithm is more complex, however, and spreads the embedded data over the image in a random way or encrypts the data before embedding, it may be nearly impossible to detect.

One method of detecting steganography in images may be to examine the colors in an image. If, in a 24 bit image, there are a great many colors which differ by only a small amount, it may indicate tampering. This theory is based on the idea that most images use a very small fraction of the 2^{24} colors available. If a steganographic technique uses the lower two bits of each byte, then any pixel could be changed by at most a factor of 2^6 . Assuming that most images use a particular color hundreds or thousands of times, there could be 2^6 variations of that color all within 2^6 values of each other. A 2^6 change in a pixel's color value is imperceptible, so a clump of colors like this would be suspicious. Of course, it is possible that the original image was created with such a fine gradation of color, but this is probably unlikely. A survey of many images from various sources could provide insight on this point.

2. How Widespread is the Use of Steganography?

If a technique or set of techniques could be devised to detect steganography, it would be interesting to conduct a survey of images available on the Internet to determine if steganography is used, by whom, and for what purposes. Steganographic applications are available on the Internet, but it is not known if they are being used. Moreover, there are probably other privately developed and used applications in existence.

3. Steganography on the World Wide Web

The World Wide Web (WWW) makes extensive use of inline images. There are literally millions of images on various web pages worldwide. It may be possible to develop an application to serve as a web browser to retrieve data embedded in web page images. This "stego-web" could operate on top of the existing WWW and be a means of covertly disseminating information.

4. Steganography in Printed Media

If data is embedded in an image, the image printed, then scanned and stored in a file, can the embedded data be recovered? This would require a special form of steganography to which could allow for inaccuracies in the printing and scanning equipment.

5. Anti-Steganography Measures

As was seen in this thesis, JPEG garbles any unencoded steganographically embedded data. Also, palettization (mapping a large number of colors in an image to a smaller subset of colors) of an image will render it unsuitable for steganography [3]. It is likely, as with JPEG, that some means may be employed to prevent loss of steganographically embedded data when its wrapper file is processed. The question remains open as to what is the most effective anti-steganography tool or set of tools.

LIST OF REFERENCES

1. Amoroso, Edward, *Fundamentals of Computer Security Technology*, PTR Prentice Hall, 1994.
2. Brassil, J., Low, S., Maxemchuk, N., O'Garman, L., "Electronic Marking and Identification Techniques to Discourage Document Copying," IEEE Infocom 94, pp 1278-1287.
3. Cha, S.D., Park, G.H., and Lee, H.K., "A Solution to the On-Line Image Downgrading Problem," in Proceedings of the Eleventh Annual Computer Security Applications Conference, New Orleans, LA, pp. 108-112, December 1995.
4. Denning, Dorothy E., "Cryptographic Checksums for Multilevel Database Security," Proceedings of the 1984 Symposium on Security and Privacy, 29 Apr - 2 May, 1984, Oakland, CA, pp. 52-61.
5. Gasser, Morrie, *Building A Secure Computer System*, Van Nostrand Reinhold, New York, 1988.
6. Joint Photographic Experts Group (JPEG) Compression for the National Imagery Transmission Format Standard, MIL-STD-188-198A, December 1993.
7. Kahn, D., *The Codebreakers*, MacMillan Company, 1967.
8. Kodak, <http://www.kodak.com:80/digitalImaging/samples/varietyPix.shtml>.
9. Kurak, C., McHugh J., "A Cautionary Note on Image Downgrading," Proceedings of the 8th Annual Computer Security Applications Conference, 1992, pp 153-159.
10. Lockstone, Keith, "A Class of Steganographic Protocols," Paper posted to the USENET newsgroup sci.crypt.research, 29 Nov. 1994.
11. Pennebaker, William B., Mitchell, Joan L., *JPEG Still Image Compression Standard*, Van Nostrand Reinhold, New York, 1993.
12. Simmons, G.J., "The Prisoner's Problem and the Subliminal Channel," Advances in Cryptology: Proceedings of Crypto 83, Plenum Press, 1984, pp. 51-67.
13. Wallace, Gregory K., "The JPEG Still Picture Compression Standard," Communications of the ACM, Vol. 34, No. 4, April 1991.
14. Walton, S., "Image Authentication for a Slippery New Age," Dr. Dobbs Journal, No. 229, April '95, pp. 18-26.

APPENDIX: SOURCE CODE

All code is written in C++ and compiled using the GNU project C++ Compiler (v2.4)

A. GENERIC STEGO EMBEDDING PROGRAM

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "stego_functions.h"

/* 21 Feb - This stego program embeds data from a textin file into a
   infile and stores the result as outfile. The data bits are
   embedded in the lower density (a user specified parameter)
   bits of each byte of the infile. */

fstream infile, // infile is the 24 bit .BMP file
          outfile, // outfile is the modified 24 bit .BMP file
          textin; // textin is the text file to be embedded in the image file

char image_file[32], steg_file[32], textsource_file[32]; // file names

unsigned char old_textbyte, hide_mask, image_byte, full_short, data_hide;

int x, // loop counter
    bits_left; // number of bits which have been read from the data file,
              // but still need to be embedded in the image file

unsigned short density, steg_head2, // density at which data is embedded, placed
                               // in header for use during extraction
              textbyte;

unsigned long text_bytes, // number of bytes in the data file
              steg_head1, // number of bytes embedded in the image, placed
                          // in header to allow for extraction
              hide, // hold lower density bits of byte to be embedded
              full_long;

bmpheader header; // This is the current infile header.
```

```

typedef struct {
    unsigned short blue, green, red; }
color;

// Function prototypes
void bitslice(unsigned short inbyte, unsigned short bitarray[8]);
void insert_1bit(color& pixel, unsigned short textbit);

main()
{
    // read in names of image file and output file
    cout << "\n";
    cout << "Enter name of the image file : ";
    cin >> image_file;
    cout << "Enter name of the textsource file : ";
    cin >> textsource_file;
    cout << "\n";
    cout << "Enter name of the output/steg file : ";
    cin >> steg_file;
    cout << "\n";

    // Open the input image file and read the header
    cout << "\n";
    infile.open(image_file, ios::in);
    read_bmp_header(infile, header);

    cout << "File_size: " << header.file_size << "\n";
    cout << "Offset bits: " << header.offset_bits << "\n";
    cout << "Size: " << header.size << "\n";
    cout << "Width: " << header.width << "\n";
    cout << "Height: " << header.height << "\n";
    cout << "Bit count: " << header.bit_count << "\n";
    cout << "Image size: " << header.image_size << "\n";
    cout << "X pixels: " << header.x_pixels << "\n";
    cout << "Y pixels: " << header.y_pixels << "\n";
    cout << "Number colors: " << header.number_colors << "\n";
    cout << "Colors important: " << header.colors_important << "\n";
    cout << "\n";

    outfile.open(steg_file, ios::out);
    // Write the header to the output file
    write_bmp_header(outfile, header);
}

```

```

// Now that the header is written, we are ready to insert the data
// bits into the image bytes as follows:
// 1. First, count how many bytes are in the data file.
// 2. Embed the data byte count in the image bytes.
// 3. Read in a data byte.
// 4. Bitslice the byte into an array of its component bits
// 5. Insert the eight data bits into the image bytes and write the bytes to
//    the output stego image file file. The number of data bits/image byte
//    is contained in the density variable.
// 6. Get another text byte

textin.open(textsource_file, ios::in);
cout << "textsource file is open.\n\n";
text_bytes = byte_count(textin);
cout << "The input text file has " << text_bytes << " bytes.\n";

// Reset the input text file by closing it then opening it.
textin.close();
textin.open(textsource_file, ios::in);

// Prompt the user for density at which data should be embedded
cout << "Enter the density you would like to use for the embedding > ";
cin >> density; // Embed density bits per byte
bits_left = 8; // Bits left to embed

// Make sure the data will fit
if ( ((text_bytes*8)/density + 25) < header.image_size) {
// Embed the number of data bytes to be hidden and the density. These
// values will constitute the stego 'header' to allow recovery of the data.
// The header will be embedded with a density of 2. (2 is arbitrary)
// So a total of 16 + 8 = 24 image bytes will be used for the stego header.
steg_head1 = text_bytes;
steg_head2 = density;
cout << "\nData length: " << steg_head1 << " Density: " << density << '\n';
cout << "\n";
for (x=0; x<16; ++x) { // loop to write the steg_head1 value
    full_long = 0;
    hide = 3;
    hide = hide & steg_head1;
    // Hide now contains lower 2 bits of steg_head1

full_long = full_long | hide;
infile.read((unsigned char*) &image_byte, 1);

```

```

image_byte = image_byte >> 2;
image_byte = image_byte << 2;
image_byte = image_byte | full_long;    // Embed bits
    outfile.write((unsigned char*) &image_byte, 1);
steg_head1 = steg_head1 >> 2;          // get rid of lower 2 bits
}
for (x=0; x<8; ++x) {    // loop to write the steg_head2 value
full_long = 0;
hide = 3;
hide = hide & steg_head2;
    // Hide now contains lower 2 bits of steg_head2

full_long = full_long | hide;
infile.read((unsigned char*) &image_byte, 1);
    image_byte = image_byte >> 2;
image_byte = image_byte << 2;
image_byte = image_byte | full_long; // Embed bits
    outfile.write((unsigned char*) &image_byte, 1);
steg_head2 = steg_head2 >> 2; // get rid of lower 2 bits
}

hide_mask = (unsigned short) pow(2, (double) density) - 1;

// Insert bits into the image bytes at the specified density
while (!textin.eof()) {
    textbyte = 0;
    textin.read((unsigned char*) &textbyte, 1); // Read a text byte
    textbyte = textbyte >> 8;
    if (bits_left < density) { // We must put the leftover bits at the end
        // of the new textbyte
        //shift left to accommodate old bits
        textbyte = textbyte << bits_left;
        textbyte = textbyte | old_textbyte; // tack on the old bits
        bits_left = bits_left + 8; // The old bits + the new bits
    }

    while (bits_left >= density) {
        full_short = 0;
        data_hide = hide_mask;
        data_hide = data_hide & textbyte;
        // Hide now contains lower density bits of textbyte

        full_short = full_short | data_hide;
    }
}

```

```

image_byte = 0;
infile.read((unsigned char*) &image_byte, 1);
image_byte = image_byte >> density; // Clear lower density bits
image_byte = image_byte << density;
image_byte = image_byte | full_short; // Embed density bits
    outfile.write((unsigned char*) &image_byte, 1);
textbyte = textbyte >> density; //get rid of lower density bits
bits_left = bits_left - density;
old_textbyte = textbyte; // save any remaining bits
// End while
// At this point there are less than density bits to embed.
// There are two possibilities: bits_left = 0 or bits_left > 0
// In any case a new textbyte must be read.
// End while

// At end of input file so just continue to write pixels without modification.
while (!infile.eof() ) {
    infile.read((unsigned char*) &image_byte, 1);
    outfile.write((unsigned char*) &image_byte, 1);
}

} // End if data will fit
else {
    cout << "Data file is too big to fit into the image.\n";
} // End else data will fit

outfile.close();
textin.close();
infile.close();

return 0; // End of main()
}

// *****
// This function takes a byte and breaks it into its component bits.
// The bits are stored in bitarray with the LSB in bitarray[0].

void bitslice(unsigned short inbyte, unsigned short bitarray[8]) {
    int i;
    for (i=0; i<8; ++i) {
        bitarray[i] = 0;
        if (inbyte & 1) {
            bitarray[i] = 1;

```

```

        inbyte >> 1;
    }
    else inbyte >> 1;
    }
} // End of bitslice()

//*****
// This function takes a pixel structure and a text bit,
// determines which is the largest color component of the
// pixel, and inserts the bit in that color component.
// It hides one bit inside one pixel.

void insert_1bit(color& pixel, unsigned short textbit) {
    unsigned short* big = &pixel.red;

    if (pixel.red >= pixel.green) big = &pixel.red;
    else {
        big = &pixel.green;
    }
    if (*big < pixel.blue) big = &pixel.blue;

    if (textbit == 1) { *big = *big | 1; } // Makes the LSB 1
    else { *big = *big >> 1;
        *big = *big << 1; } // Makes the LSB 0
    } // End of function insert_1bit()

```

B. GENERIC STEGO EXTRACTION PROGRAM

```
/*      Stego Extraction Program
```

```
21Feb - This version of Extract will extract data hidden at any density (1-7).  
The user will be prompted for the density. If 0 is entered, the  
program will try to find the stego header to get the density and  
the number of hidden bytes. */
```

```
#include <iostream.h>  
#include <fstream.h>  
#include <math.h>  
#include "stego_functions.h"  
#define PI 3.14159265
```

```
fstream infile, // infile is the 24 bit .BMP file with embedded text  
outfile; // outfile is the extracted text file
```

```
char steg_file[32], data_out_file[32]; // file names
```

```
unsigned char stegin, mask, hidden_byte, inbyte, upper1, upper2, upper3,  
             upper4, orphan_bits, densin;
```

```
int x; // loop counter
```

```
unsigned short density, leftover_bits, bytes_recovered;
```

```
unsigned long steg_bytes;
```

```
bmpheader header; // This is the current infile header.
```

```
typedef struct {  
    unsigned short blue, green, red; }  
color;
```

```
// Function prototypes
```

```
void bitslice(unsigned short inbyte, unsigned short bitarray[8]);
```

```
void insert_1bit(color& pixel, unsigned short textbit);
```

```
main()
```

```
{  
    // Get the name of the input image file containing stego data (steg_file)  
    // and the name of the file in which the extracted data will be stored.  
    cout << "\n";
```

```

cout << "Enter the stego image file name: ";
cin >> steg_file;
cout << "Enter a file name for the extracted data: ";
cin >> data_out_file;

infile.open(steg_file, ios::in);
read_bmp_header(infile, header);

cout << "\nStego image file information: \n\n";
cout << "File_size: " << header.file_size << '\n';
cout << "Offset bits: " << header.offset_bits << '\n';
cout << "Size: " << header.size << '\n';
cout << "Width: " << header.width << '\n';
cout << "Height: " << header.height << '\n';
cout << "Bit count: " << header.bit_count << '\n';
cout << "Image size: " << header.image_size << '\n';
cout << "X pixels: " << header.x_pixels << '\n';
cout << "Y pixels: " << header.y_pixels << '\n';
cout << "Number colors: " << header.number_colors << '\n';
cout << "Colors important: " << header.colors_important << '\n';

infile.close();
infile.open(steg_file, ios::in);
// We have reset the stego file and now are ready to read the
// stego header.

// Now we must extract the stego header consisting of the number of bytes
// of hidden data and the density at which it was hidden.
// Get the number of bytes...
steg_bytes = 0; // steg_bytes will contain the number of hidden bytes
for (x=0; x<16; ++x) {
    infile.seekg((69 -x), ios::beg);
    steg_bytes = steg_bytes << 2; // Make room for the next 2 bits
    infile.read((unsigned char*) &stegin, 1);
    stegin = stegin & 3; // Get rid of all bits above the lower two.
    steg_bytes = steg_bytes | stegin; // Put the bits on the bottom of
    // steg_bytes
}
// steg_bytes now contains the number of hidden bytes in the image.
// Now get the density....
density = 0;
for (x=0; x<8; ++x) {

```



```
hidden_byte = hidden_byte | (inbyte << 4);
infile.read((unsigned char*) &inbyte, 1);
inbyte = inbyte & mask;
hidden_byte = hidden_byte | inbyte;
outfile.write((unsigned char*) &hidden_byte, 1);
++bytes_recovered;
```

case 5: // Density 5555555555555555555555555555555555

```
mask = 31;
hidden_byte = 0;
switch (leftover_bits) {
```

case 0:

```
infile.read((unsigned char*) &inbyte, 1);
inbyte = inbyte & mask;
    // Store the lower 5 bits
hidden_byte = hidden_byte | inbyte;
    // Get another byte
infile.read((unsigned char*) &inbyte, 1);
    // Store the upper 3 bits
upper3 = inbyte & 7;
hidden_byte = hidden_byte | (upper3 << 5);
orphan_bits = (inbyte & 24) >> 3;
// Orphan bits are bits 3,4
leftover_bits = 2;
outfile.write((unsigned char*) &hidden_byte, 1);
++bytes_recovered;
break;
```

case 1:

```
hidden_byte = orphan_bits; //Low bit now stored
infile.read((unsigned char*) &inbyte, 1);
inbyte = inbyte & mask;
    // Store the bits 1,2,3,4,5
hidden_byte = hidden_byte | (inbyte << 1);
infile.read((unsigned char*) &inbyte, 1);
upper2 = inbyte & 3;
    // Store the bits 6,7
hidden_byte = hidden_byte | (upper2 << 6);
orphan_bits = (inbyte & 28) >> 2;
// Orphan bits are bits 2,3,4. Right justify
leftover_bits = 3;
outfile.write((unsigned char*) &hidden_byte, 1);
++bytes_recovered;
```

```

break;

case 2:
    hidden_byte = orphan_bits;
        // Low 2 bits now stored
    infile.read((unsigned char*) &inbyte, 1);
    inbyte = inbyte & mask;
        // Store the bits 2,3,4,5,6
    hidden_byte = hidden_byte | (inbyte << 2);
    infile.read((unsigned char*) &inbyte, 1);
    upper1 = inbyte & 1;
        // Store the bit 7
    hidden_byte = hidden_byte | (upper1 << 7);
        // Pull out orphan bits and right justify
    orphan_bits = (inbyte & 30) >> 1;
    leftover_bits = 4;
    outfile.write((unsigned char*) &hidden_byte, 1);
    ++bytes_recovered;
    break;

case 3:
    hidden_byte = orphan_bits;
        // Low 3 bits now stored
    infile.read((unsigned char*) &inbyte, 1);
    inbyte = inbyte & mask;
        // Store the bits 3,4,5,6,7
    hidden_byte = hidden_byte | (inbyte << 3);
    leftover_bits = 0;
    outfile.write((unsigned char*) &hidden_byte, 1);
    ++bytes_recovered;
    break;

case 4:
    hidden_byte = orphan_bits;
        // Low 4 bits now stored
    infile.read((unsigned char*) &inbyte, 1);
    upper4 = inbyte & 15;
        // Store the bits 2,3,4,5,6
    hidden_byte = hidden_byte | (upper4 << 4);
        //pull out orphan bit and right justify it
    orphan_bits = (inbyte & 16) >> 4;
    leftover_bits = 1;
    outfile.write((unsigned char*) &hidden_byte, 1);

```



```

infile.read((unsigned char*) &inbyte, 1);
upper1 = inbyte & 1;
        // Store the upper bit
hidden_byte = hidden_byte | (upper1 << 7);
orphan_bits = (inbyte & 254) >> 1;
leftover_bits = 7;
outfile.write((unsigned char*) &hidden_byte, 1);
++bytes_recovered;
break;

    case 2:
hidden_byte = orphan_bits >> 1;
        // Low bit now stored
infile.read((unsigned char*) &inbyte, 1);
inbyte = inbyte & mask;
        // Store the bits 2,3,4
hidden_byte = hidden_byte | (inbyte << 2);
infile.read((unsigned char*) &inbyte, 1);
inbyte = inbyte & mask;
        // Store the bits 5,6,7
hidden_byte = hidden_byte | (inbyte << 5);
leftover_bits = 0;
outfile.write((unsigned char*) &hidden_byte, 1);
++bytes_recovered;
break;

    default: cout << "\n Invalid leftover. Aborting!\n";
return 0;

} // End of leftover switch
break;

default: cout << "Invalid density!  Aborting.\n";
return 0;

} // End density switch

if (infile.eof()) return 0; // Don't let it go beyond the end of the file.
} // End while bytes_recovered

return 0;
}

```



```

// *****
// This function takes a byte and breaks it into its component bits.
// The bits are stored in bitarray with the LSB in bitarray[0].

void bitslice(unsigned short inbyte, unsigned short bitarray[8]) {
    int i;

    for (i=0; i<8; ++i) {
        bitarray[i] = 0;
        if (inbyte & 1) {
            bitarray[i] = 1;
            inbyte >> 1;
        }
        else inbyte >> 1;
    }
    } // End of bitslice()

//*****
// This function takes a pixel structure and a text bit,
// determines which is the largest color component of the
// pixel, and inserts the bit in that color component.
// It hides one bit inside one pixel.

void insert_1bit(color& pixel, unsigned short textbit) {
    unsigned short* big = &pixel.red;

    if (pixel.red >= pixel.green) big = &pixel.red;
    else {
        big = &pixel.green;
    }
    if (*big < pixel.blue) big = &pixel.blue;

    if (textbit == 1) { *big = *big | 1; } // Makes the LSB 1
    else { *big = *big >> 1;
        *big = *big << 1; } // Makes the LSB 0

    } // End of function insert_1bit()

```

C. IMAGE FILE COMPARISON PROGRAM

```
#include <iostream.h>
#include <fstream.h>
#include "stego_functions.h"

/* 21 Feb 96 - This program compares 2 image files, file1 and file2, and finds
the Hamming distance between each pixel. The 2 files must be
exactly the same size. It produces an error map file which
displays the Hamming distances as color coded pixels.
This program is used to look at the errors caused by JPEG
or by a steganographic program. */

fstream file1,file2, // 24 bit .BMP files to be compared
mapfile;           // 24 bit .BMP error map file

char file_1[32], file_2[32], errormap[32]; // file names

unsigned char f1byte_b, f1byte_g, f1byte_r, f2byte_b, f2byte_g, f2byte_r,
// blue, green and red pixel values for file1 and file2
max = 0xff, min = 0x00, mid = 0x0f, white = 0xff;
// color channel values

int x, y, // loop counters
error_pixels, // number of overall pixels which differ
one_3_diff = 0, four_6_diff = 0, seven_9_diff = 0, ten_12_diff = 0,
thirteen_15_diff = 0, sixteen_18_diff = 0, nineteen_21_diff = 0,
twentytwo_24_diff = 0;
// number of pixels which differ by these numbers of bits

unsigned short diff_b, diff_r, diff_g, // number of bits differing in each
// color channel
intermediate_diff, total_diff; // total number of bits differing in
// each pixel

bmpheader header1,header2;

// Function prototypes
unsigned short bits_different(unsigned char byte1, unsigned char byte2);

main()
{
// read in names of file1, file2, and the errormap file
cout << "\n";
```

```

cout << "Enter name of file #1 : ";
cin >> file_1;
cout << "Enter name of file #2 : ";
cin >> file_2;
cout << "Enter a filename for the errormap : ";
cin >> errormap;
cout << "\n";

// Open file1 and read the header
file1.open(file_1, ios::in);
read_bmp_header(file1, header1);

cout << "File # 1 : " << file_1 << '\n';
cout << "File_size: " << header1.file_size << '\n';
cout << "Offset bits: " << header1.offset_bits << '\n';
cout << "Size: " << header1.size << '\n';
cout << "Width: " << header1.width << '\n';
cout << "Height: " << header1.height << '\n';
cout << "Bit count: " << header1.bit_count << '\n';
cout << "Image size: " << header1.image_size << '\n';
cout << "X pixels: " << header1.x_pixels << '\n';
cout << "Y pixels: " << header1.y_pixels << '\n';
cout << "Number colors: " << header1.number_colors << '\n';
cout << "Colors important: " << header1.colors_important << '\n';
cout << "\n";

// Open file2 and read the header
file2.open(file_2, ios::in);
read_bmp_header(file2, header2);

cout << "File # 2 : " << file_2 << '\n';
cout << "File_size: " << header2.file_size << '\n';
cout << "Offset bits: " << header2.offset_bits << '\n';
cout << "Size: " << header2.size << '\n';
cout << "Width: " << header2.width << '\n';
cout << "Height: " << header2.height << '\n';
cout << "Bit count: " << header2.bit_count << '\n';
cout << "Image size: " << header2.image_size << '\n';
cout << "X pixels: " << header2.x_pixels << '\n';
cout << "Y pixels: " << header2.y_pixels << '\n';
cout << "Number colors: " << header2.number_colors << '\n';
cout << "Colors important: " << header2.colors_important << '\n';
cout << "\n";

```

```

/* We are now at the beginning of the pixels */
// Check to see if the number of pixel bytes is the same.
if (header1.image_size == header2.image_size) {
// They are equal so the comparison can be made

mapfile.open(errormap, ios::out);
// Write the header to the output file
write_bmp_header(mapfile, header1);

error_pixels = 0;
for (y = 0; y < header1.image_size; y=y+3)
{
file1.read((unsigned char*) &f1byte_b, 1);
file2.read((unsigned char*) &f2byte_b, 1);
file1.read((unsigned char*) &f1byte_g, 1);
file2.read((unsigned char*) &f2byte_g, 1);
file1.read((unsigned char*) &f1byte_r, 1);
file2.read((unsigned char*) &f2byte_r, 1);

if ( (f1byte_b != f2byte_b) || (f1byte_g != f2byte_g) || (f1byte_r != f2byte_r) )
{
++error_pixels;
diff_b = bits_different(f1byte_b, f2byte_b);
diff_g = bits_different(f1byte_g, f2byte_g);
diff_r = bits_different(f1byte_r, f2byte_r);
intermediate_diff = (diff_b + diff_g + diff_r);
if (intermediate_diff <= 1)
total_diff = 0;
else
total_diff = (intermediate_diff - 1) / 3;
switch (total_diff)
{
// 1, 2, or 3 bits differ
case 0 : mapfile.write((unsigned char*) &min, 1);
mapfile.write((unsigned char*) &min, 1);
mapfile.write((unsigned char*) &max, 1);
++one_3_diff;
break;
// 4, 5, or 6 bits differ
case 1 : mapfile.write((unsigned char*) &min, 1);
mapfile.write((unsigned char*) &max, 1);
mapfile.write((unsigned char*) &max, 1);

```

```

++four_6_diff;
    break;
    // 7, 8, or 9 bits differ
    case 2 : mapfile.write((unsigned char*) &max, 1);
mapfile.write((unsigned char*) &min, 1);
mapfile.write((unsigned char*) &max, 1);
++seven_9_diff;
    break;
    // 10, 11, or 12 bits differ
case 3 : mapfile.write((unsigned char*) &min, 1);
mapfile.write((unsigned char*) &max, 1);
mapfile.write((unsigned char*) &min, 1);
++ten_12_diff;
    break;
    // 13, 14, or 15 bits differ
case 4 : mapfile.write((unsigned char*) &max, 1);
mapfile.write((unsigned char*) &max, 1);
mapfile.write((unsigned char*) &min, 1);
++thirteen_15_diff;
    break;
    // 16, 17, or 18 bits differ
case 5 : mapfile.write((unsigned char*) &max, 1);
mapfile.write((unsigned char*) &min, 1);
mapfile.write((unsigned char*) &min, 1);
++sixteen_18_diff;
    break;
    // 19, 20, or 21 bits differ
case 6 : mapfile.write((unsigned char*) &mid, 1);
mapfile.write((unsigned char*) &mid, 1);
mapfile.write((unsigned char*) &mid, 1);
++nineteen_21_diff;
    break;
    // 22, 23, or 24 bits differ
case 7 : mapfile.write((unsigned char*) &min, 1);
mapfile.write((unsigned char*) &min, 1);
mapfile.write((unsigned char*) &min, 1);
++twentytwo_24_diff;
    break;
default : cout << "default - bad value in switch statement\n";
} // end switch
} // end if
else
{

```

```

        mapfile.write((unsigned char*) &white, 1);
mapfile.write((unsigned char*) &white, 1);
mapfile.write((unsigned char*) &white, 1);
    }

    } // end for y
    } // The image_size if statement

else {
cout << "\nThe bitmaps are of different size. \n";
}

cout << "There are " << error_pixels << " pixels which are different.\n";
cout << one_3_diff << " differ by 1, 2, or 3 bits.\n";
cout << four_6_diff << " differ by 4, 5, or 6 bits.\n";
cout << seven_9_diff << " differ by 7,8, or 9 bits.\n";
cout << ten_12_diff << " differ by 10, 11, or 12 bits.\n";
cout << thirteen_15_diff << " differ by 13, 14, or 15 bits.\n";
cout << sixteen_18_diff << " differ by 16, 17, or 18 bits.\n";
cout << nineteen_21_diff << " differ by 19, 20, or 21 bits.\n";
cout << twentytwo_24_diff << " differ by 22, 23, or 24 bits.\n";
cout << "\n";

mapfile.close();
file1.close();
file2.close();

return 0;

} // End main()

// *****
// determine whether bits differ

unsigned short bits_different(unsigned char byte1, unsigned char byte2) {
unsigned char xor_byte = 0;
    unsigned short count = 0;

xor_byte = byte1 ^ byte2;
for ( x=0; x <8; ++x) {
count +=(xor_byte & 1);
xor_byte = xor_byte >> 1; }
return count;}

```

D. RGB VECTOR ENCODING STEGO PROGRAM

/* 19 Mar 96 - This version embeds using the modulo-distance-from-origin technique and repeats each message bit several times.

Some variables are defined in the stego_functions.h file.

*/

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "stego_functions.h"
```

```
fstream infile, // infile is the 24 bit .bmp file
        outfile, // outfile is the modified 24 bit .bmp file
        textin; // textin is the text to be hidden
```

```
char image_file[32], steg_file[32], textsource_file[32]; // variables to hold
        // user-supplied file names
```

```
unsigned char c, r, g, b,
        textbyte,
        raw_red, raw_green, raw_blue, // image file bytes
        nothing; // unused byte
```

```
int x, y, z, i, // loop variables
    d_mod, // vector length modulo steg_modulo
    epsilon, // amount of displacement of the pixel along its line to origin
    repeat,
    skip; // number of pixels to skip between protocol events
```

```
long total_pixels, // number of pixels in the image
    pixelnum; // number of pixels used to embed data
```

```
double dist, // distance to the origin from the pixel
    t, // parametric equation variable for computing the pixel's
        // new position
    red, green, blue; // the integer versions of the color channels
```

```
unsigned long text_bytes; // number of bytes in the text file
```

```
bmpheader header; // the infile header.
```

```

// Function prototype
void in_bounds(double& red, double& green, double& blue);

main()
{
    // read in names of image file and output file
    cout << "\n";
    cout << "Enter name of the image file : ";
    cin >> image_file;

    cout << "\nEnter name of the textsource file : ";
    cin >> textsource_file;
    cout << "\n";
    cout << "Enter name of the output/steg file : ";
    cin >> steg_file;
    cout << "\n";

    // Open the input image file and read the header
    cout << "\n";
    infile.open(image_file, ios::in);
    read_bmp_header(infile, header);

    // Print the image file's header data
    cout << "File_size: " << header.file_size << '\n';
    cout << "Offset bits: " << header.offset_bits << '\n';
    cout << "Size: " << header.size << '\n';
    cout << "Width: " << header.width << '\n';
    cout << "Height: " << header.height << '\n';
    cout << "Bit count: " << header.bit_count << '\n';
    cout << "Image size: " << header.image_size << '\n';
    cout << "X pixels: " << header.x_pixels << '\n';
    cout << "Y pixels: " << header.y_pixels << '\n';
    cout << "Number colors: " << header.number_colors << '\n';
    cout << "Colors important: " << header.colors_important << '\n';
    cout << "\n";

    // Open the output image file and write its header
    outfile.open(steg_file, ios::out);
    write_bmp_header(outfile, header);
}

```



```

// Open the input text file and count the number of characters in it
textin.open(textsource_file, ios::in);
text_bytes = byte_count(textin);
cout << "The input text file has " << text_bytes << " bytes.\n";
textin.close();
// Reset the text file
textin.open(textsource_file, ios::in);

// Insert bits into the image pixels.
infile.seekg(header.offset_bits, ios::beg); // Make sure we are past the
// file header.

// Make sure data will fit
total_pixels = header.width * header.height;
if (text_bytes*8*redundancy > total_pixels) {
    cout << "Input data file is too big! Aborting. \n";
    return 0;
}

// Compute skip factor. skip is the number of pixels to skip
skip = (header.width*header.height) / ((text_bytes*8)*redundancy);
skip = skip - 3;
if (skip < 0) skip = 0;
cout << "Skip = " << skip << "\n";

pixelnum = 0;

while (!textin.eof() ) { //loop until the text file is empty

    textin.read((unsigned char*) &textbyte, 1); // Read a text byte

    for (x=0; x<8; ++x) { //Embed all 8 bits of the text byte

        for (repeat=0; repeat<redundancy; ++repeat) { // Embed each bit
            // #redundancy times

            // Read the 3 bytes of a pixel
            infile.read((unsigned char*) &raw_red, 1);
            infile.read((unsigned char*) &raw_green, 1);
            infile.read((unsigned char*) &raw_blue, 1);
            ++pixelnum;
        }
    }
}

```

```

// Convert the values to integers
red = raw_red;
green = raw_green;
blue = raw_blue;

// Make sure no value is clear of the upper and lower bounds
in_bounds(red, green, blue);

// Compute the distance from the origin
dist = sqrt((double)(red*red + green*green + blue*blue));
d_mod = ((int) dist) % steg_modulo;

// Find out if the pixel needs to slide and how much.
// Set the values of the raw color variables to the
// value which will be written to the stego image file.

if (textbyte & 1) { // Embed a 1

    if (d_mod == encode_1) epsilon = 0;
    else ;
    if (d_mod < encode_1 && d_mod >= encode_0) epsilon = encode_1 - d_mod;
    else ;
    if (d_mod > encode_1) epsilon = -1*(d_mod - encode_1);
    else ;
    if (d_mod < encode_0) epsilon = -1*(lo_to_hi + d_mod);
    else ;

    // t is the variable in the parametric equations for
    // the line on which the pixel lies.
    t = ((double) epsilon)/dist;
    // Compute the new values for the pixels
    red = red + red*t;
    green = green + green*t;
    blue = blue + blue*t;
} // End if

else { // Embed a 0

    if (d_mod == encode_0) epsilon = 0;
    else ;

```

```

if (d_mod > encode_0 && d_mod <= encode_1) epsilon = -1*(d_mod - encode_0);
else ;
if (d_mod > encode_1) epsilon = (encode_0 + (steg_modulo - d_mod));
else ;
if (d_mod < encode_0) epsilon = (encode_0 - d_mod);
else ;

// t is the variable in the parametric equations for
// the line on which the pixel lies.
t = ((double) epsilon)/dist;
// Compute the new values for the pixels
red = red + red*t;
green = green + green*t;
blue = blue + blue*t;

} // end else

raw_red = (unsigned char) red; // Real value is truncated on conversion
raw_green = (unsigned char) green;
raw_blue = (unsigned char) blue;

outfile.write ((unsigned char*) &raw_red, 1);
outfile.write ((unsigned char*) &raw_green, 1);
outfile.write ((unsigned char*) &raw_blue, 1);

} // End of 'repeat redundancy times' loop

// Skip some pixels between embedded bits
for (z=0; z<skip; ++z) {

    infile.read((unsigned char*) &raw_red, 1);
    infile.read((unsigned char*) &raw_green, 1);
    infile.read((unsigned char*) &raw_blue, 1);
    ++pixelnum;

    outfile.write ((unsigned char*) &raw_red, 1);
    outfile.write ((unsigned char*) &raw_green, 1);
    outfile.write ((unsigned char*) &raw_blue, 1);
} // for z

textbyte >>= 1;

```

```

    } // End for x to embed 8 bits

    } // end while ! textin.eof()

    .

// At end of input file so just continue to write pixels without modification.
while (!infile.eof() ) {
    infile.read((unsigned char*) &raw_red, 1);
    infile.read((unsigned char*) &raw_green, 1);
    infile.read((unsigned char*) &raw_blue, 1);
    if (!infile.eof()) {
        outfile.write ((unsigned char*) &raw_red, 1);
        outfile.write ((unsigned char*) &raw_green, 1);
        outfile.write ((unsigned char*) &raw_blue, 1); }
}

cout << "Pixels used: " << pixelnum << '\n';

outfile.close();
textin.close();
infile.close();

return 0; // End of main()

}

void in_bounds(double& red, double& green, double& blue) {

    // This function checks to make sure the color values are
    // in bounds. If not the value is adjusted to be in bounds.

    if (red > hi_bound) red = hi_bound;
    else {
        if (red < lo_bound) red = lo_bound; }

    if (green > hi_bound) green = hi_bound;
    else {

```

```
    if (green < lo_bound) green = lo_bound; }  
  
    if (blue > hi_bound) blue = hi_bound;  
    else {  
        if (blue < lo_bound) blue = lo_bound; }  
  
} // end in_bounds
```

E. RGB VECTOR STEGO EXTRACTION PROGRAM

/*

This version of extract will extract data hidden by modifying the modulus of the distance from the origin. This version uses a user supplied skip factor.

Some variables are defined in the stego_functions.h file.

*/

```
#include <iostream.h>
```

```
#include <fstream.h>
```

```
#include <math.h>
```

```
#include "stego_functions.h"
```

```
fstream infile, // infile is the 24 bit .bmp file
```

```
outfile; // outfile is the modified 24 bit .bmp file
```

```
char steg_file[32], data_out_file[32]; // variables for the user-supplied  
// file names
```

```
unsigned char hidden_byte, // the byte of data being recovered  
raw_red, raw_green, raw_blue, // image file bytes  
nothing; // unused byte
```

```
unsigned char c, r, g, b;
```

```
int x,y,z,i, // loop variables
```

```
red, green, blue, // integer versions of the color values
```

```
d_mod, // distance to origin modulo steg_modulo
```

```
skip; // number of pixels to skip between protocol events
```

```
double dist;
```

```
unsigned short bytes_recovered; // count of bytes recovered so far
```

```
unsigned long hidden_bit; // holds the bits representing the hidden bit
```

```
long pixelnum; // number of pixels used in the recovery process
```

```
unsigned long steg_bytes=0;
```

```
bmpheader header; // This is the current infile header.
```

```
// Function prototype
```

```
int byteweight(unsigned long hbyte); // Counts the number of 1's in an unsigned  
// long variable
```

```

main()
{
    // Get the name of the input image file containing stego data (steg_file)
    // and the name of the file in which the extracted data will be stored.
    cout << "\n";
    cout << "Enter the stego image file name: ";
    cin >> steg_file;
    cout << "Enter a file name for the extracted data: ";
    // cin >> data_out_file;
    data_out_file = "steg.out";
    cout << "\nThe data will be extracted based on mod62\n";
    cout << "How many bytes are hidden? ";
    cin >> steg_bytes;
    cout << '\n';
    // cout << "What is the skip factor? ";
    // cin >> skip;
    skip = 17;
    cout << '\n';

    infile.open(steg_file, ios::in);
    read_bmp_header(infile, header);

    cout << "\nStego image file information: \n\n";
    cout << "File_size: " << header.file_size << '\n';
    cout << "Offset bits: " << header.offset_bits << '\n';
    cout << "Size: " << header.size << '\n';
    cout << "Width: " << header.width << '\n';
    cout << "Height: " << header.height << '\n';
    cout << "Bit count: " << header.bit_count << '\n';
    cout << "Image size: " << header.image_size << '\n';
    cout << "X pixels: " << header.x_pixels << '\n';
    cout << "Y pixels: " << header.y_pixels << '\n';
    cout << "Number colors: " << header.number_colors << '\n';
    cout << "Colors important: " << header.colors_important << '\n';
    cout << '\n';

    infile.close();
    infile.open(steg_file, ios::in);

    outfile.open(data_out_file, ios::out); // Open the file that will
    // hold the extracted data.

```

```

// We need to be at the first byte of the first pixel.
// The image header is contained in the
// first 54 bytes so we go to the 55th byte which contains the
// least significant bit(s) of the first byte of the hidden data.
// The seekg value is an offset from the first byte of the file.

infile.seekg(header.offset_bits, ios::beg);

// Begin a loop to read all the bytes containing data.

bytes_recovered = 0;
pixelnum = 0;

while (bytes_recovered < steg_bytes) {

    hidden_byte = 0;
    for (x=0; x<8; ++x) { // Loop 8 times to recover 1 byte of hidden data
        hidden_bit = 0;
        for (y=0; y<redundancy; ++y) { // Loop redundancy # times to get one bit
            // Read a pixel
            infile.read((unsigned char*) &raw_red, 1);
            infile.read((unsigned char*) &raw_green, 1);
            infile.read((unsigned char*) &raw_blue, 1);
            ++pixelnum;

            // Convert the values to integers
            red = (int) raw_red;
            green = (int) raw_green;
            blue = (int) raw_blue;

            // Compute the distance from the origin
            dist = sqrt((double)(red*red + green*green + blue*blue));
            d_mod = ((int) dist) % steg_modulo;

            if (d_mod > pivot) hidden_bit |= (1 << y);
            else;
        } // For y

        // Skip some pixels after each bit is read
        for (z=0; z<skip; ++z) {

```



```

        infile.read((unsigned char*) &nothing, 1);
        infile.read((unsigned char*) &nothing, 1);
        infile.read((unsigned char*) &nothing, 1);
        ++pixelnum;

    } // for z

    if (byteweight(hidden_bit) > (int)(redundancy/2)) hidden_byte |= (1 << x);
    else;

    } // End of x loop
    outfile.write((unsigned char*) &hidden_byte, 1);
    ++bytes_recovered;

    if (infile.eof()) return 0; // Don't let it go beyond the end of the file.
    } // End while bytes_recovered
    cout << "Pixels used: " << pixelnum << '\n';

    infile.close();
    outfile.close();

    return 0;

}
//*****

int byteweight(unsigned long hbyte) {

    int x, w;
    w=0;
    for (x=0; x<15; ++x) {
        if (hbyte & 1) ++w;
        hbyte >>= 1;
    }
    return w;
}

```

F. STEGO FUNCTIONS HEADER FILE

```
// This is the stego_functions.h file

#include <fstream.h>

struct bmpheader {          // This structure holds the 24 bit .bmp
    char bm[2];             // header information. There is no palette
    unsigned long file_size; // data.
    unsigned short reserved[2];
    unsigned long
    offset_bits,
    size,
    width,
    height;
    unsigned short
    planes,
    bit_count;
    unsigned long
    compression,
    image_size,
    x_pixels,
    y_pixels,
    number_colors,
    colors_important;
};

int redundancy = 5; // the number of times to embed a bit

// Set the modulo parameters
int steg_modulo = 62,
    pivot = 30, // Anything over 30 is a 1
    encode_1 = 47,
    encode_0 = 16,
    hi_bound = 224,
    lo_bound = 31,
    lo_to_hi = 15;

/*int steg_modulo = 42, commented out since we are using steg_modulo=62
pivot = 20, // Anything over 20 is a 1
encode_1 = 31,
encode_0 = 10,
hi_bound = 235,
lo_bound = 20,
```

```

lo_to_hi = 11;*/

// Function prototypes
void read_bmp_header(fstream& infile, bmpheader& header);
void write_bmp_header(fstream& outfile, bmpheader& header);
long int byte_count(fstream& intext);
unsigned long read_lsbfirst_long(fstream& infile);
unsigned short read_lsbfirst_short(fstream& infile);
void write_lsbfirst(fstream& outfile, unsigned long value);
void write_lsbshortfirst(fstream& outfile, unsigned short value);

// The .bmp file format was designed for dos based machines.
// The values of the header are stored in little endian format.
// The stego programs were developed on big endian workstations.
// The functions below read and write these little endian
// values to and from big endian variables.

/*****
// This function reads a .bmp header into a bmpheader structure
// which is passed by reference.
void read_bmp_header(fstream& infile, bmpheader& header) {
    infile.read((unsigned char*) &header.bm[0], 1);
    infile.read((unsigned char*) &header.bm[1], 1);
    header.file_size = read_lsbfirst_long(infile);
    header.reserved[0] = read_lsbfirst_short(infile);
    header.reserved[1] = read_lsbfirst_short(infile);
    header.offset_bits = read_lsbfirst_long(infile);
    header.size = read_lsbfirst_long(infile);
    header.width = read_lsbfirst_long(infile);
    header.height = read_lsbfirst_long(infile);
    header.planes = read_lsbfirst_short(infile);
    header.bit_count = read_lsbfirst_short(infile);
    header.compression = read_lsbfirst_long(infile);
    header.image_size = read_lsbfirst_long(infile);
    header.x_pixels = read_lsbfirst_long(infile);
    header.y_pixels = read_lsbfirst_long(infile);
    header.number_colors = read_lsbfirst_long(infile);
    header.colors_important = read_lsbfirst_long(infile);
} // End of read_bmp_header()

/*****
// This function writes a .bmp header to a file.

```

```

void write_bmp_header(fstream& outfile, bmpheader& header) {
    outfile.write((unsigned char*) &header.bm[0], 1);
    outfile.write((unsigned char*) &header.bm[1], sizeof(header.bm[1]));
    write_lsblongfirst(outfile, header.file_size);
    write_lsbsshortfirst(outfile, header.reserved[0]);
    write_lsbsshortfirst(outfile, header.reserved[1]);
    write_lsblongfirst(outfile, header.offset_bits);
    write_lsblongfirst(outfile, header.size);
    write_lsblongfirst(outfile, header.width);
    write_lsblongfirst(outfile, header.height);
    write_lsbsshortfirst(outfile, header.planes);
    write_lsbsshortfirst(outfile, header.bit_count);
    write_lsblongfirst(outfile, header.compression);
    write_lsblongfirst(outfile, header.image_size);
    write_lsblongfirst(outfile, header.x_pixels);
    write_lsblongfirst(outfile, header.y_pixels);
    write_lsblongfirst(outfile, header.number_colors);
    write_lsblongfirst(outfile, header.colors_important);
} // End of write_bmp_header()

// *****
long int byte_count(fstream& intext) {
    // This function counts the number of bytes in a file and
    // returns the count as a long integer.
    int count = 0;

    while (!intext.eof()) {
        (void) intext.get();
        ++count;
    }
    return count;
} // End of byte_count

// *****
// Read a little endian long integer

unsigned long read_lsbfirst_long(fstream& infile) {

    unsigned char buffer[4];
    unsigned long value;

    infile.read((unsigned char*) &buffer, 4);
    value = (unsigned long) (buffer[3] << 24);

```

```

value |= (unsigned long) (buffer[2] << 16);
value |= (unsigned long) (buffer[1] << 8);
value |= (unsigned long) (buffer[0]);

return value;

} // End of unsigned long

// *****
// Read a little endian short integer

unsigned short read_lsbfirst_short(fstream& infile) {

    unsigned char buffer[2];
    unsigned short value;

    infile.read((unsigned char*) &buffer, 2);
    value = (unsigned short) (buffer[1] << 8);
    value |= (unsigned short) (buffer[0]);

    return value;

} // End of unsigned short

/*infile.seekg(-1, ios::cur); Back up one byte /
infile.write((unsigned char*) &new_c, 1); */

// *****
// Write a little endian long integer

void write_lsbfirst(fstream& outfile, unsigned long value) {
    unsigned char
        buffer[4];

    buffer[0] = (unsigned char) (value);
    buffer[1] = (unsigned char) ((value) >> 8);
    buffer[2] = (unsigned char) ((value) >> 16);
    buffer[3] = (unsigned char) ((value) >> 24);
    outfile.write((unsigned char*) &buffer, 4);

```

```
} // End of unsigned long
```

```
// *****
```

```
// Write a little endian short integer
```

```
void write_lsshortfirst(fstream& outfile, unsigned short value) {  
    unsigned char  
        buffer[2];
```

```
    buffer[0] = (unsigned char) (value);  
    buffer[1] = (unsigned char) ((value) >> 8);  
    outfile.write((unsigned char*) &buffer, 2);  
} // End of unsigned short
```

```
//  
*****
```

INITIAL DISTRIBUTION LIST

- | | | |
|----|---|----|
| 1. | Defense Technical Information Center
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, VA 22060-6218 | 2 |
| 2. | Dudley Knox Library
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943-5101 | 2 |
| 3. | Chairman, Code CS
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943 | 2 |
| 4. | Raymond Isbell
Central Imagery Office
8401 Old Courthouse Road
Vienna, VA 22182-3280 | 2 |
| 5. | Dr. Blaine Burnham
National Security Agency
Research and Development Building
R23
9800 Savage Road
Fort Meade, MD 20755-6000 | 2 |
| 6. | William Marshall
National Security Agency
Research and Development Building
R23
9800 Savage Road
Fort Meade, MD 20755-6000 | 2 |
| 7. | Dr. Cynthia E. Irvine
Computer Science Department
Code CS/Ic
Naval Postgraduate School Monterey, CA 93943-5118 | 15 |

- | | | |
|-----|--|---|
| 8. | Dr. Harold Fredricksen
Mathematics Department, Code MA/FS
Naval Postgraduate School
Monterey, CA 93943-5118 | 5 |
| 9. | Commanding Officer
Fleet Information Warfare Center
2555 Amphibious Drive
NAB Little Creek
Norfolk, VA 23521-3225 | 2 |
| 10. | Naval Information Warfare Activity
ATTN: CDR J. O'Dwyer
9800 Savage Road
Fort Meade, MD 20755-6000 | 2 |
| 11. | LT Hannelore Campbell
USS Boxer (LHD 4)
FPO AP 96661 | 3 |
| 12. | LT Daniel L. Currie III
Fleet Information Warfare Center
2555 Amphibious Drive
NAB Little Creek
Norfolk, VA 23521-3225 | 5 |